(12) **UK Patent Application** (19) **GB** (11) **2 256 514** (13) **A**

(43) Date of A publication 09.12.1992

(21) Application No 9210876.0

(22) Date of filing 21.05.1992

(30) Priority data
(31) 07703394 (32) 21.05.1991 (33) US

(71) Applicant
Digital Equipment Corporation

(Incorporated in the USA – Massachusetts)

146 Main Street, Maynard, Massachusetts 01754,
United States of America

(72) Inventor
Yoav Raz

(74) Agent and/or Address for Service
Kilburn & Strode
30 John Street, London, WC1N 2DD, United Kingdom

(51) INT CL⁵
G06F 9/46 15/16

(52) UK CL (Edition K)
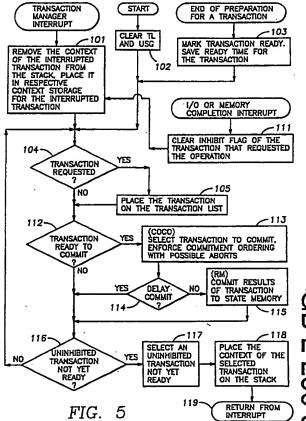G4A AMB

(56) Documents cited
EP 0457112 A2 US 4881166 A US 4249241 A

(58) Field of search
UK CL (Edition K) G4A AMB AUD
INT CL⁵ G06F 9/46 15/16

(54) **Ordered processing of transactions**

(57) Serializability across distributed transactions is guaranteed by selectively committing and aborting or delaying transactions to enforce an order of commitment that is the same as an order of performance of conflicting component operations of the transactions. First and second memory access operations, for example, conflict when they reference the same memory location and at least one of them is a write operation. A computer system may permit a second transaction to read data written by a write operation of a first transaction before the first transaction is committed. In this case, depending on the respective order in which the two conflicting operations occur, the order of commitment is enforced, 113, possibly by aborting either of the two transactions. The conflicts, for example, are detected when addresses are determined during preparation of the transactions. The operations may be scheduled for most efficient use of the system capabilities. In a multiprocessor system in which a global coordinator communicates with a plurality of transaction processors by way of "prepare" and "commit" commands, acknowledgment that a transaction has been "prepared" is delayed until the transaction's "abort set" has been minimized.

FIG. 5

POWER ON

READ SWITCH FROM
NON—VOLATILE
MEMORY
51

COPY
BANK (SWITCH)
TO
BANK (SWITCH)
52

MODIFY DATA IN
BANK (SWITCH)
53

READY
TO COMMIT
?
54 NO

YES

SWITCH ← SWITCH
55

WRITE SWITCH TO
NON—VOLATILE
MEMORY
56

FIG. 2A

FIG. 2B

POWER ON

READ SWITCH FROM
NON-VOLATILE MEMORY 61

SWITCH.
SET
? 62

YES

NO

COPY BANK(1)
SAVE RECORDS
FROM BANK(1)
TO BANK(0) 63

CLEAR SWITCH IN
NON-VOLATILE
MEMORY 64

READ RECORDS FROM
BANK(0) INTO SCRATCH
MEMORY 65

MODIFY RECORDS IN
SCRATCH MEMORY 66

READY
TO COMMIT
? 67

NO

YES

COPY ORIGINAL DATA OF
RECORDS TO BE MODIFIED
FROM BANK(0) TO BANK(1) 68

SET SWITCH IN
NON-VOLATILE
MEMORY 69

WRITE MODIFIED DATA
INTO BANK(0) 70

CASE 1: T_2 CAN READ THE WRITE DATA OF
T_1 ONLY AFTER T_1 IS COMMITED

(A) $\quad T_1$ ——— $(W_X\ C_1)$ ——— — — —
$\quad\quad T_2$ ———————————— R_X — C_2 —

(B) $\quad T_1$ ——— $(W_X\ C_1)$ — — — — — $\Big\}$ INCONSISTENT
$\quad\quad T_2$ — R_X ————————— C_2 —

(C) $\quad T_1$ ——— $(W_X\ C_1)$ — — — —
$\quad\quad T_2$ — R_X — C_2 ————————

$$FIG.\quad 3A$$

CASE 2: T_2 CAN READ THE WRITE DATA OF
T_1 BEFORE T_1 IS COMMITED

(A) $\quad T_1$ —— W_X ——— C_1 —— — —
$\quad\quad T_2$ ————— R_X ——— C_2 — —

(B) $\quad T_1$ ————— W_X — C_1 — — — $\Big\}$ INCONSISTENT
$\quad\quad T_2$ — R_X ————————— C_2 —— —

(C) $\quad T_1$ ————— W_X ——— C_1 — —
$\quad\quad T_2$ —— R_X ——— C_2 — — —

(D) $\quad T_1$ —— W_X ————— C_1 — — $\Big\}$ INCONSISTENT
$\quad\quad T_2$ ———— R_X — C_2 — — —

(E) $\quad T_1$ ———— W_X — C_1 — — — —
$\quad\quad T_2$ — R_X — C_2 — — — — —

(F) $\quad T_1$ — W_X — C_1 — — — —
$\quad\quad T_2$ ————— R_X — C_2 — —

$$FIG.\quad 3B$$

DIGITAL COMPUTER — 20

| TRANSACTION MANAGER (TM) — 82 | COMMITMENT ORDER COORDINATOR (COCO) — 85 | RESOURCE MANAGER (RM) — 81 |
| TRANSACTION LIST (TL) — 83 | UNDICIDED TRANSACTIONS SERIALIZABILITY GRAPH (USG) — 84 | STATE MEMORY (SM) — 28,29 |

FIG. 4A

FIG. 4B

90

DIGITAL COMPUTER 1 — 91

95 — TM   COCO   RM
94 — TL   USG   SM

DIGITAL COMPUTER 2 — 92

96 — TM   COCO   RM
TL   USG   SM

DIGITAL COMPUTER 3 — 93

97 — TM   COCO   RM
TL   USG   SM

TRANSACTION
MANAGER
INTERRUPT

START

END OF PREPARATION
FOR A TRANSACTION

101

CLEAR TL
AND USG

103

MARK TRANSACTION READY.
SAVE READY TIME FOR
THE TRANSACTION

REMOVE THE CONTEXT
OF THE INTERRUPTED
TRANSACTION FROM
THE STACK, PLACE IT
IN RESPECTIVE
CONTEXT STORAGE
FOR THE INTERRUPTED
TRANSACTION

102

I/O OR MEMORY
COMPLETION INTERRUPT

111

CLEAR INHIBIT FLAG OF THE
TRANSACTION THAT REQUESTED
THE OPERATION

104

TRANSACTION
REQUESTED
?

YES

NO

105

PLACE THE TRANSACTION
ON THE TRANSACTION LIST

112

TRANSACTION
READY TO
COMMIT
?

YES

NO

113

(COCO)
SELECT TRANSACTION TO COMMIT,
ENFORCE COMMITMENT ORDERING
WITH POSSIBLE ABORTS

YES

DELAY
COMMIT
?

NO

(RM)
COMMIT RESULTS
OF TRANSACTION
TO STATE MEMORY

114

115

116

UNINHIBITED
TRANSACTION
NOT YET
READY
?

NO

YES

117

SELECT AN
UNINHIBITED
TRANSACTION
NOT YET
READY

118

PLACE THE
CONTEXT OF THE
SELECTED
TRANSACTION
ON THE STACK

119

RETURN FROM
INTERRUPT

FIG. 5

TRANSACTION ID	POINTER TO NEXT	V	R	I	G	P
0						
1						
2						
3						
4						
· · ·						
X						

HEAD POINTER — 108

TAIL POINTER — 109

POINTER TO TRANSACTION BEING PERFORMED — 110

83

FIG. 6

	0	1	2	3	4	· · ·	X
0	0	1	0	0	0		0
1	0	0	1	0	0		0
2	0	0	0	0	0		0
3	0	1	0	0	0		0
4	0		1	1	0		0
· · ·							
X	0	0	0	0	0		0

84

FIG. 7

FIG. 8

$C_{\overline{WR}}$ ⟶
C_{WR} ⟹

FIG. 12

SELECT A READY
TRANSACTION TO
BE COMMITED —141

ABORT THE TRANSACTIONS
FOR WHICH COMMITMENT
IS CONTRARY TO THE
COMMITMENT ORDER AND
COMMITMENT OF THE
SELECTED TRANSACTION —142

CONTINUE TO COMMIT
SELECTED TRANSACTION

FIG. 9

CASCADING ABORT
OF T_X

FIND ALL T_Y SUCH THAT
T_Y HAS READ DATA
WRITTEN BY T_X —191

ABORT T_X —192

CALL CASCADING ABORT
FOR EACH T_Y —193

RETURN

FIG. 13

```
        ┌────────────────────────────────────┐        150
        │   SELECT  TRANSACTION  TO  COMMIT   │       ⟋
        └────────────────────────────────────┘
                          │                            151
        ┌────────────────────────────────────────────┐⟋
        │ GET FIRST READY TRANSACTION AND CLEAR COMMIT FLAG │
        └────────────────────────────────────────────┘
                          │                            152
        ┌──────────────────────────────────┐⟋
        │ COUNT THE NUMBER OF MEMBERS       │
        │ IN THE TRANSACTIONS ABORT SET     │
        └──────────────────────────────────┘
                          │
      153⟍            ╱ABORT╲      YES    ┌──────────────┐
                     ╱ SET NULL ╲───────▶│ CONTINUE TO  │
                     ╲    ?     ╱         │ COMMIT THE   │
                      ╲       ╱           │ TRANSACTION  │
                          │ NO            └──────────────┘
                                                 157
        ┌──────────────────────────────────────────────┐⟋
        │ COMPUTE DELAY = CURRENT TIME - READY TIME      │
        └──────────────────────────────────────────────┘
                          │
      158⟍          ╱  IS  ╲      YES    ┌──────────────┐ 159
                    ╱ DELAY > ╲──────────▶│ SET COMMIT   │⟋
                    ╲ LIMIT  ╱            │ FLAG         │
                     ╲  ?  ╱             └──────────────┘
                          │ NO                  │
   ┌──────────┐156  ◀─────┴──────────────────────┘
   │ GET NEXT │⟋
   │ READY    │       160⟍   ╱ ANY ╲
   │TRANSACTION│      YES   ╱ MORE READY ╲
   └──────────┘  ◀─────────╲ TRANSACTIONS ╱
                            ╲    ?    ╱
                                │ NO
   ┌──────────┐    NO    161⟍╱COMMIT╲  YES   ┌──────────────┐162
   │ CONTINUE │◀──────────╲FLAG SET╱────────▶│ SELECT THE READY│⟋
   │ WITHOUT  │            ╲  ?   ╱          │ TRANSACTION WITH│
   │ COMMIT   │                              │ THE MINIMUM No. │
   └──────────┘                              │ OF MEMBERS IN   │
                                             │ ITS ABORT SET   │
                                             └──────────────┘
                                                    │
                                             ┌──────────────┐
                                             │ CONTINUE TO COMMIT THE│
                                             │ SELECTED TRANSACTION  │
                                             └──────────────┘
```

FIG. 10

FIG. 11

FIG. 14

[FROM FIG. 10]

[FROM FIG. 10]

(CONTINUING WITHOUT COMMIT)

(CONTINUING WITH COMMIT)

211
TERMINATE DELAY SIGNAL RECEIVED FROM COORDINATOR ?

212
GLOBAL TRANSACTION READY ? — YES

NO

NO

YES

213
GLOBAL TRANSACTION WITH P SET IN ABORT SET ?

(CONTINUE IN THE TRANSACTION MANAGER [FIG. 5])

NO

216
SEND PREPARE ACKNOWLEDGE SIGNAL TO COORDINATOR, SET P FLAG

YES ← *214* GLOBAL TRANSACTION ?

NO

215
ABORT THE TRANSACTIONS IN THE ABORT SET AND COMMIT THE RESULTS OF THE TRANSACTION TO STATE MEMORY

FIG. 15

COMMIT/ABORT
INTERRUPT FROM
COORDINATOR

221

ABORT
?

YES → ABORT GLOBAL
TRANSACTION   222

NO

ABORT THE TRANSACTIONS
IN THE ABORT SET AND
COMMIT THE RESULTS OF
THE TRANSACTION TO STATE
MEMORY

224

SEND ACKNOWLEDGEMENT
TO COORDINATOR

223

RETURN
FROM
INTERRUPT

220

FIG. 16

254

TRANSACTION
MANAGER
(TM)

252

255

COMMITMENT
ORDER
COORDINATOR
(COCO)

251

256

RESOURCE
MANAGER
(RM)

253

FIG. 17

FIG. 18

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates generally to distributed

5   computing, and more particularly to a transaction processing

system in which component operations in related transactions are

distributed so that at least one operation in a second

transaction is performed before a first transaction having a

conflicting operation is committed. The present invention

10 specifically concerns a method and apparatus for scheduling the

performance of the conflicting operations according to available

resources and ensuring that the results of the conflicting

operations are committed in the same order as the order of

performance of the conflicting operations.

15

2. Description of the Background Art

A desirable feature of a computing system is the ability to

recover from partial system failures that interrupt memory write

operations. If an application program has a memory write

20 operation in progress at the time of the system failure, it is

most likely that the memory record will become erroneous. To

enable the recovery of memory records after a partial system

failure, it is necessary for the application program to keep

backup copies of the records in nonvolatile memory. When the

25 computing system is restarted, the memory records to be recovered

are replaced with the backup copies.

To facilitate the making of backup copies and the recovery
of memory records, the operating system typically provides an
established set of memory management procedures that can be
invoked or called from an application program to define a

5   "recovery unit." The recovery unit consists of program
statements between a "START" statement and a "COMMIT" statement.
All of the statements in the "recovery unit" must be completed
before the memory records modified by the statements in the
recovery unit are made available for subsequent processing. The

10   "START" statement corresponds to the making of a backup copy in
nonvolatile memory, and the "COMMIT" statement corresponds to
switching of the backup copy with a modified version. The
statements in the "recovery unit" specify operations in a single
"transaction." Upon recovering from a partial system error,

15   inspection of the nonvolatile memory will reveal that the
operations in the single "transaction" are either all completed,
or none of them are completed.

In a distributed computing system, the operations in a
single transaction may modify files in different data bases, and

20   the files may be shared by other processes. During the operation
of the transaction, the files may be inconsistent for a time,
although the files will be consistent upon completion of the
transaction. A typical example is a transfer of funds from one
account to another, in which a first account is debited, and at a

25   slightly later time, another account is credited. During the
interim, the two accounts are inconsistent because the sum of the

two accounts does not represent the total funds in the two
accounts. Due to inconsistency when files are being modified by
a transaction, it is known to prevent other processes from
accessing the files until the modification is finished.

5   Recoverability can be assured in this example by performing
commitment for both files at the same time and place. By
changing a single flag, for example, the backup copies of each
file can be replaced at the same time with the modified versions
of the files. In many instances, however, it is desirable to

10 distribute the operations in a transaction among multiple
processors or processes in a computing system, and to commit the
transaction by committing the operations in each process or
processor while permitting some variability between the times of
commitment. In these instances, an "atomic commitment protocol"

15 is typically used to ensure recoverability. The protocol
requires the exchange of information about the state of the
transaction between the processors or processes. To identify the
transaction being performed, the transaction is typically
assigned a unique "transaction identification number."

20   A widely used atomic commitment protocol is known as the
"two-phase commit protocol." In a somewhat elementary example of
this protocol, one processor or process in the computing system
is assigned the role of a coordinator which initiates a
transaction. To begin a transaction, the coordinator sends a

25 prepare command to all of the processors or processes
participating in the transaction.

Upon receipt of the "prepare" command, each processor or process participating in the transaction performs a "START" operation by first placing "write locks" on memory accessed by the transaction, writes the transaction identification number

5 into permanent memory to remember that it is prepared for the transaction, and then sends an acknowledgement back to the coordinator processor, but does not yet perform its part of the transaction. The coordinator waits for acknowledgements from all of the participants. When the coordinator receives

10 acknowledgements from all of the participants, the coordinator records in permanent memory a list of the participants and a notation that the transaction is now being completed, and then the coordinator sends "commit" commands to all of the participants. The coordinator, however, may receive a message

15 from a participant indicating that it cannot prepare for the transaction, or the coordinator may fail to receive acknowledgements from all of the participants after a predetermined time period, possibly after the coordinator has retransmitted the "prepare" command. In this case the

20 coordinator transmits an "abort" command to all of the participants.

Upon receipt of the "commit" command, each participant checks its permanent memory for the transaction identification number to determine whether the participant has prepared for the

25 transaction, and if it has, it performs its part of the transaction, and then performs a "COMMIT" operation to update the

state of permanent memory and clear the transaction ID from

permanent memory in one "atomic" step, and erase the write locks.

Then the participant sends an acknowledgement back to the

coordinator. When the coordinator receives acknowledgments from

5 all of the participants, it erases the list of participants from

permanent memory, and the transaction is finished.

In a many distributed computing systems, the processors or

processes are permitted to perform multiple transactions

simultaneously. In the usual case each processor or process

10 performs transactions that are local to the processor or process,

and also performs portions of global transactions. In a

distributed data base system, for example, local data base

queries and edits may occur locally, and some of the

modifications may be made globally. A direct application of the

15 two-phase commit protocol described above may perform

satisfactorily in such a system, so long as the global

transactions can be given a high priority with respect to the

local transactions. But use of the read and write locks may

unnecessarily restrict local transactions that could be processed

20 concurrently.

Additional complexity is introduced when it is desired to

process global transactions concurrently across multiple

processors or processes in a distributed computing system. It is

impractical to permit a processor or process to view a global

25 picture of all the conflicts in all of the other processors or

processes. Without a global picture, however, it is difficult

for a processor or process to ensure that there is a correlation between its seriablility order and the seriability orders of the other processors or processes. Time-stamping of transaction requests and data updates is one method that has been used to

5   address this problem of concurrency control. In general, concurrency control in a distributed computing system has been achieved at the expense of restricted autonomy of the local processors or processes, or by locking.

The problem of global deadlock also has to be addressed

10 whenever global transactions are performed concurrently. One known solution is to provide a global transaction manager that decides whether or not to dispatch concurrent global transaction requests. An example is described Y. Breitbart et al., "Reliable Transaction Management in a Multidatabase System", <u>Proc. of the</u>

15 <u>ACM SIGMOD conf. on Management of Data</u>, Atlantic City, New Jersey, June 1990, pp. 215-224. The global scheduler keeps track of global transaction requests for local locks on data items by using a global lock mechanism. Each global data item has a global lock associated with it. A global transaction that needs

20 only to read a data item requests a global read-lock. Locks are conflicting if they are requested by two different transactions on the same data item and at least one of the requested locks is a write-lock. If two global transactions request conflicting global locks, the scheduler will prevent one of the transactions

25 from proceeding because it knows that the two transactions will cause a conflict at the local site. The scheduler uses strict

two-phase locking for allocating global locks to global

transactions, and maintains a global "wait for graph." The

"global wait for graph" is a directed graph $G=(V,E)$ whose set of

vertices V is a set of global transactions and an edge $T_i \rightarrow T_j$

5   belongs to E if and only if global transaction T_i waits for a

global lock allocated to global transaction T_j. If a global

transaction waits for a global lock, then the transaction state

becomes "blocked" and the transaction is included in the "global

wait for graph." The transaction becomes active again only after

10   it can obtain global locks that it was waiting for. To avoid

global deadlocks, the "global wait for graph" is always made

acyclic. To ensure data consistency in the presence of failures,

the scheduler also uses a "commit graph" and a "wait-for-commit

graph" to determine when to schedule a commit operation. The

15   commit graph $CG=<TS,E>$ is an undirected bipartite graph whose set

of nodes TS consists of a set of global transactions (transaction

nodes) and a set of local sites (site nodes). Edges from E may

connect only transaction nodes with site nodes. An edge (T_i,S_j)

is in E if and only if transaction T_i was executing at site S_j,

20   and the commit operation for T_i has been scheduled for

processing. After the commit operation for T_i is completed, T_i

is removed from the commit graph along with all edges incidental

to T_i. Global database consistency is assured if the commit

graph does not contain any loops. The wait-for-commit graph is a

25   directed graph $G=(V,E)$ whose set of vertices V consists of a set

of global transactions. An edge $T_i \rightarrow T_j$ is in E if and only if T_i

has finished its execution, but its commit operation is still pending and T_j is a transaction whose commit operation should be completed or aborted before the commit of T_i can be scheduled. The scheduler uses the following algorithm for constructing the

5 wait-for-commit graph, and in scheduling a commit operation of transaction T_i:

1. For each site S_k in which T_i is executing, temporarily add the edge $T_i \rightarrow S_k$ to the commit graph.

2. If the augmented commit graph does not contain a cycle,

10 then the global commit operation is submitted for processing, and the temporary edges become permanent.

3. If the augmented commit graph contains a cycle then:

a) The edges $T_i \rightarrow T_{i1}, \ldots, T_i \rightarrow T_{im}$ are inserted into the wait-for-commit graph. The set $\{T_{i1}, T_{i2}, \ldots, T_{im}\}$

15 consists of all the transactions which appear in the cycle which was created as a result of adding the new edges to the commit graph.

b) Remove the temporary edges from the commit graph. The transaction T_i, however, need not necessarily wait for the

20 completion of every transaction T_{ik} such that $T_i \rightarrow T_{ik}$. It may be ready to be scheduled for a commit operation after some of transactions T_{ik} such that $T_i \rightarrow T_{i1}$ ($0<l<r$) successfully commit (and in some cases, a successful commit of only one such transaction would be sufficient to schedule the transaction's commit!).

SUMMARY OF THE INVENTION

The present invention guarantees serializability across distributed transactions in a computing system by selectively committing and aborting the transactions to enforce an order of

5   commitment that is the same as an order of performance of conflicting component operations of the transactions. When the transaction is committed, results of the component operations are committed to state memory. When the operation is aborted, the results of the component operations are discarded. A first

10   memory access operation in a first transaction, for example, conflicts with a second memory access operation in a second transaction when the two memory access operations reference the same memory location and at least one of the operations is a write operation.

15   In a typical prior art transaction processing system, a second transaction can read data written by a first transaction only after the second transaction has been committed. This restriction is a sufficient condition to ensure recoverability of the system. To practice the present invention this case, when a

20   second transaction performs a read operation before a conflicting write operation of a first transaction is committed at a time when the second transaction has not yet committed, the second transaction is aborted to ensure that the order in which the transactions are committed is not different from the order in

25   which the conflicting operations are performed.

The present invention, however, permits the construction of
a transaction processing system in which a second transaction may
read data written by a write operation of a first transaction
before the first transaction is committed. In this case,

5   depending on the respective order in which the two conflicting
operations occur, either of the two transactions may be aborted
to ensure that the order of commitment is the same as the order
of performance of the conflicting operations. Moreover, to
insure recoverability, both of the transactions should be aborted

10   in the case of the read operation following the write operation
and the read operation being performed before aborting of the
write operation. In general, in a transaction processing system
in which a second transaction may read data written by a write
operation of a first transaction, recoverability is enforced by a

15   process of cascading aborts; the aborting of a transaction
requires the additional aborting of all other transactions that
have read data written by aborted transactions.

In cases where memory addresses of memory access operations
are known prior to preparing the transactions, the required

20   commitment order may be determined prior to preparation of the
transactions. Otherwise, conflicts are detected when the memory
addresses are determined during preparation of the transactions.

The commitment order is enforced by committing a selected
transaction for which a result has been prepared, and aborting

25   other transactions for which a result is being prepared or is
prepared and for which commitment is contrary to the predefined

commitment order and the committing of the selected transaction.
The transaction to commit is selected, for example, by comparing
priorities assigned to the transactions, by an ordering of the
transactions in a list, by a commitment request from a

5   coordinator, or by a strategy to minimize the number of other
transactions that are aborted as a result of the selection.   In
a multiprocessor system in which a global coordinator
communicates with a plurality transaction processors by way of
"prepare" and "commit" commands, the minimization strategy

10   preferably is used to delay acknowledging that a transaction has
been "prepared" until the transaction's "abort set" has been
minimized.

BRIEF DESCRIPTION OF THE DRAWINGS

15   A more detailed understanding of the invention may be had
from the following description of a preferred embodiment given by
way of example and to be understood in conjunction with the
accompanying drawings in which:

FIG. 1 is a block diagram of a digital computer configured

20   for transaction processing;

FIG. 2A is a flow chart of a procedure for performing
transaction processing in the computer of FIG. 1 by switching
between two banks of state memory;

FIG. 2B is an alternative procedure for operating the

25   Digital Computer of FIG. 1 for transaction processing by saving

·copies of only the data records of state memory that are modified by a transaction;

FIG. 3A illustrates various scheduling possibilities for conflicting memory access operations of distributed transactions

5   for the case in which a second transaction can read the write data of a first transaction only after the first transaction is committed;

FIG. 3B illustrates various scheduling possibilities for conflicting memory access operations of distributed transactions

10   for the case in which a second transaction can read the write data of a first transaction before the first transaction is committed;

FIG. 4A shows a digital computer configured in accordance with a preferred embodiment of the present invention to enforce a

15   commitment ordering in which distributed transactions are committed in the order in which conflicting component operations are performed;

FIG. 4B illustrates a distributed computing system including a plurality of the digital computers as shown in FIG. 4A;

20   FIG. 5 illustrates a scheduling procedure employed by a transaction scheduler component of the digital computer FIG. 4A;

FIG. 6 illustrates an organization of a transaction list and related pointers which are used by the transaction scheduler for scheduling the performance of component operations of distributed

25   transactions;

FIG. 7 is a schematic diagram illustrating a data structure corresponding to a graph of commitment ordering between distributed transactions having conflicting component operations;

FIG. 8 is a pictorial diagram of the graph corresponding to

5 the data stored in the data structure of FIG. 7;

FIG. 9 is a flow chart of a procedure that references the data structure of FIG. 7 to enforce the commitment order;

FIG. 10 is a flow chart of a procedure for selecting a transaction to commit;

10 FIG. 11 is a procedure for detecting a conflicting memory access operation during the preparation of a transaction;

FIG. 12 is a modified graph in which write-read conflicts are distinguished from other conflicts;

FIG. 13 is a flow chart of a recursive procedure for

15 insuring recoverability by performing cascading aborts;

FIG. 14 is a state diagram of the digital computer shown in FIG. 4A when used in a multi-processing system of FIG. 4B for processing both local and global transactions;

FIG. 15 is a flow chart of commitment scheduling procedures

20 that handle global transactions in a different fashion than local transactions;

FIG. 16 is a flow chart of a procedure for committing and aborting transactions in response to signals from a coordinator of a global transaction;

25 FIG. 17 is a block diagram showing a commitment order coordinator employing the present invention inserted in a

conventional transaction processing system between a transaction manager and a resource mananger; and

FIG. 18 is a state diagram of the transaction processing system of FIG. 17.

5   While the invention is susceptible to various modifications and alternative forms, specific embodiments thereof have been shown by way of example in the drawings and will be described in detail herein. It should be understood, however, that it is not intended to limit the invention to the particular forms

10   disclosed, but on the contrary, the intention is to cover all modifications, equivalents, and alternatives falling within the spirit and scope of the invention as defined by the appended claims.

15   DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

Turning now to FIG. 1, there is shown a block diagram generally designated 20 of a digital computer configured for transaction processing. The computer 20 includes a central processing unit 21 for executing programmed instructions; a

20   volatile random access memory 22 for holding instructions or data; a non-volatile memory 23 such as a hard disk drive, an input/output unit 24, and a real time clock 25. The non-volatile memory 23 includes a program memory 26 in which programs are stored, and a scratch memory area 27 for storing data records.

25   Typically, the digital computer 20 executes programs which have been transferred from the program memory 26 to the volatile

random access memory 22. During the execution of a program, it
is often necessary to operate upon an amount of data that exceeds
the capacity of the volatile random access memory 22. In this
case, data records are alternately stored and retrieved from the
5   scratch memory area 27.

A common problem associated with the digital computer 20 is
the likelihood that the execution of instructions by the central
processing unit will become disrupted due to a hardware failure,
software error or power failure. A power failure, for example,
10   will cause the disappearance of data and programs stored in the
volatile random access memory 22. The problem of the loss of
data in the volatile random access memory 22 due to a power
failure can be solved by storing back-up copies of data in the
non-volatile memory 23. The back-up copies, however, must be
15   made in such a way that considers the possibility of failure
during a write operation to the non-volatile memory 23. In this
case the data record affected by the write operation might have
been corrupted and therefore must be discarded.

To deal with the problem of possible failure when writing to
20   non-volatile memory, there has been established a method of
programming called "transaction processing" which guarantees that
a portion of the non-volatile memory (referred to hereinafter as
"state memory") will either be unaffected by a transaction or
will be properly updated by results of a transaction, in the
25   presence of the failures. Transaction processing is based upon
the technique of making a back-up copy of state memory before the

results of a transaction are written to state memory, and also

writing in non-volatile memory an indication of either a first

processing phase in which the back-up copy is being made, or a

second processing phase in which the results of a transaction are

5   being written to state memory, in order to indicate which copy

might have been corrupted during a failure. For making a back-up

copy of state memory, for example, the non-volatile memory 23

includes two banks of state memory 28 and 29. To provide an

indication of which bank of stat memory might have been corrupted

10   by a failure, the non-volatile memory 23 includes a memory

location 30 for storing a switch or flag.

When recovering from a failure, it is desirable to know the

transaction that was last performed by the central processing

unit 21, so that processing can be resumed from the interrupted

15   point without repeating or skipping a transaction. For this

purpose, whenever the state memory in either of the memory banks

28 or 29 is updated, a transaction identification code 31, 32 is

written into the state memory along with the time 33, 34 at which

the results of the transaction were first written (i.e.,

20   committed) to state memory.

Turning now to FIG. 2A, there is shown a flow chart of a

procedure for guaranteeing that when recovering from a failure,

the state memory of the computer 20 shown in FIG. 1 is either

unaffected by a transaction or is properly updated by the result

25   of a transaction. Assume, for example, that the computer system

is turned on after a power failure. In a first step 51, the

central processing unit 21 reads the value of the switch 30

stored in the non-volatile memory 23. This switch indicates

which of the two banks of state memory 28, 29 might possibly have

been corrupted by the power failure. In step 52, the central

5   processing unit 21 references the value of the switch to read the

bank of state memory known not to have been corrupted, and to

make a "working copy" of the data in the other bank of state

memory. Therefore, after step 52, both bank 28 and bank 29 of

state memory have the same contents. Moreover, the transaction

10   identifier 31, 32 and the commit time 33, 34 can be inspected to

find the location in a program of the next transaction to be

processed.

In step 53 that processing is continued by modifying the

data in the working copy of state memory by writing results of

15   the transaction being processed. The end of processing of the

transaction is reached in step 54. To commit the results of the

transaction to state memory, the value of the switch is changed

in step 55, and in step 56 the changed value of the switch is

written into the switch location 30 of the non-volatile memory.

20   When viewed by the central processing unit 21 during recovery

from a failure, the writing of the changed value of the switch

into the non-volatile memory has the effect of insuring that the

committed result of the transaction either has no effect upon the

state memory, or properly updates the state memory, depending

25   upon whether the failure occurs before or after the value of the

switch has been written into the non-volatile memory. Because

the value of the switch 30 is a single bit and the switch 30 is stored in a record different from the records of the banks of state memory, any failure occurring during the writing of the this single bit is inconsequential; in this case, neither of the

5   banks of state memory should be corrupted, so the value of the switch does not matter.

The method of committing the result of a transaction as illustrated in FIG. 2A is rather inefficient when the result of a transaction modifies only a small portion of the state memory.

10   In this case, step 52 spends a good deal of time unnecessarily copying data records that have not been modified. This unnecessary copying can be eliminated by the somewhat more complex procedure of FIG. 2B.

In the first step 61 of FIG. 2B, the switch is read from the

15   non-volatile memory. Next, in step 62, the central processing unit checks whether the switch is set. If so, then a failure occurred during the processing phase in which the results of a transaction were being committed to state memory, as further described below. Therefore, in step 63 records saved in the

20   state memory bank 29 are copied to state memory bank 28. Then, in step 64, the switch in non-volatile memory is cleared.

To process a transaction, in step 65, data records are read from the state memory bank 28 and transferred into the scratch memory area 27. Then in step 66 the records in scratch memory

25   are modified in accordance with results of the transaction. When the transaction is finished, as found in step 67, then in step

68, original data of records to be modified are copied from state memory bank 28 to the state memory bank 29. Then in step 69 the switch is set in non-volatile memory. Then in step 70 the results of the transaction are committed by writing the modified

5   data into the state memory bank 28. Finally, in step 64, the switch is cleared in non-volatile memory. Processing of the next transaction begins in step 65.

As described above with respect to Fig. 2A or Fig. 2B, it is assumed that transactions are processed in sequence by the

10 central processing unit 21 of the computer 20 in FIG. 1. The present invention, however, permits the processing of transactions to be distributed in such a way that the results for a second transaction may be prepared before the results of a first transaction are committed. As will be further described

15 below with reference to FIG. 5, the present invention permits the use of a transaction scheduler incorporating a real-time operating system or scheduler to more effectively keep the central processing unit busy during the time that it would otherwise be waiting for the completion of input/output or memory

20 access operations. By permitting a second transaction to perform component operations before a first transaction has committed its results, however, introduces a problem of inconsistency unless the scheduling of operations and commitment of the transactions conforms to a certain commitment ordering. In particular, the

25 transactions should be committed in the same order as the order

in which respective conflicting (i.e. non-commuting) operations are performed.

When the scheduling of component operations and the commitment of transactions has this property of "commitment ordering", it can also be shown that in a distributed processing system (as further described below in connection with FIG. 4B), global serializability is guaranteed when only "atomic commitment" is used to coordinate the various processors in the system. This is demonstrated by a rather elaborate mathematical proof, which is appended to the present specification. From a practical standpoint, this result means that the advantages of the present invention can be applied to a conventional distributed transaction processing system by modifying each transaction processor or node in the system without modification of the global scheduler or the protocol for distributing global transactions through the system.

A conventional transaction processing system insures that a second transaction can read the write data of a first transaction only after the first transaction is committed. This is a sufficient but not necessary condition to insure recoverability. In a first embodiment of the present invention, this condition can also be maintained to minimize the amount of non-volatile memory required and to avoid what is known as "cascading aborts" to achieve recoverability. In this first embodiment, transactions are essentially processed as shown in FIG. 2B wherein the read operations read records from the bank 28 of

state memory and transfer them to scratch memory as shown in step

65 of FIG. 2B. The write operations are prepared by modifying

records in memory as shown in step 66 of FIG. 2B, but the results

of each transaction are kept in separate records in scratch

5   memory 27, even though the write operations of different

transactions reference the same records in state memory. This is

done because the effect that such conflicting write operations

will have upon the state memory is determined by the order in

which the transactions are committed.

10   FIG. 3A shows three different possibilities for the

scheduling of a first transaction having a write operation and a

second transaction having a conflicting read operation. In

general, two operations are conflicting when they are memory

access operations that access the same location in state memory

15   and at least one of the operations is a write operation. By

inspection it can be seen that of the three scheduling

possibilities, the possibility (b) violates the commitment

ordering requirement and therefore may cause inconsistency in the

state of the state memory. Due to the fact that the write

20   operation W_x does not commute with the read operation R_x, the

result for the transaction T_2 for the scheduling possibility (b)

may be different from the result for the transaction T_2 for the

scheduling possibility (a). To obtain consistent results, the

present invention permits conflicting operations of two

25   transactions to be scheduled in a selected order to most

efficiently use resources available to the central processing

unit, but insures consistency by enforcing a commitment order
that is the same as the order in which the conflicting operations
are performed.   Inconsistent scheduling possibilities, such as
the possibility (b) in FIG. 3A, are prohibited by aborting a

5   conflicting transaction when a selected transaction is committed,
or by delaying commitment of a selected transaction until after
the conflicting transaction is committed.

In the example of FIG 3A, for example, suppose that the
first operation scheduled is a read operation R_x of the second

10   transaction T_2, as shown in possibilities (b) and (c). If the
transaction T_2 is committed before the transaction T_1 as shown in
possibility (c), no inconsistency will result because the
scheduling is in conformance with commitment ordering. If,
however, the first transaction T_1 is committed before the second

15   transaction T_2 as shown in possibility (b), then the second
transaction T_2 must be aborted because otherwise commitment of
the second transaction T_2 would violate commitment ordering and
may lead to inconsistent results.

The present invention further permits the scheduling of

20   operations such that a second transaction T_2 can read the write
data of a first transaction T_1 before the first transaction T_1 is
committed. In this case recoverability can be guaranteed by a
process of cascading aborts, as further described below with
reference to FIGS. 12 and 13. For the case of a first

25   transaction T_1 having a write operation W_x and a second
transaction T_2 having a conflicting read operation R_x, there are

six scheduling possibilities, denoted in FIG. 3B as (a) to (f).
Two of these scheduling possibilities (b) and (d) violate
commitment ordering and therefore may lead to inconsistent
results. The present invention prevents these scheduling

5   possibilities from occurring by determining the order in which
the conflicting operations are performed and then delaying
commitment of a selected transaction or aborting a conflicting
operation if necessary to enforce commitment ordering. To permit
the read operation in the second transaction to read the write

10   data of a write operation in a first transaction, the digital
computer 20 of FIG. 1, for example, maintains a working copy of
state memory in the scratch memory area 27. Whenever a write
operation is performed, the results of the write operation are
written to the working copy of state memory in the scratch memory

15   area, and also separate copies of the written records are kept in
scratch memory in association with each transaction. When the
transaction is committed, only then are the results of the write
operation of the transaction written into state memory. In the
embodiment of Case 2 the read operations reference the working

20   copy of state memory in scratch memory, instead of the state
memory itself as in Case 1, so that in Case 2 the read operations
read the write data of prior write operations.

Turning now to FIG. 4A, there is shown a block diagram of
the programming and data structures used in the digital computer

25   20 of FIG 1. for scheduling transactions and enforcing commitment
ordering. To commit the results of transactions to state memory

28, 29 and to recover from failures, the digital computer is

provided with a resource manager (RM) 81 that, for example,

performs the operations shown in FIG 2B. In general, a resource

manager (RM) is a software component that manages state memory

5   resources affected by committing transactions in such a way that

the memory state of the resources can be restored before the

transaction is committed by effectively undoing all of the

changes introduced by the transaction. In other words, the

resource manager ensures that the transactions have the property

10   of "atomicity", or "all or nothing" semantics upon its state

memory resources. A resource is typically, but not necessarily,

a data item or a data object. Examples of resource managers are

typically found in data base systems (DSB's), list managers, and

cache managers.

15   To schedule the component operations of transactions in the

most efficient order based on available resources of the

computer, there is provided a transaction scheduler (TS) 82.

Preferably the transaction scheduler 82 includes a kind of real-

time operating system that manages a transaction list (TL) 83 as

20   further described below with reference to FIG 5. Preferably the

presence of conflicting operations is detected in real time when

the transactions are performed, as further described below with

reference to FIG. 11. In accordance with commitment ordering,

the order in which conflicting operations are performed

25   determines a required order of commitment of transactions, and

this required order of commitment is recorded in an undecided

transactions serializability graph (USG) 84 which is a data
structure in memory, and which is described further below with
reference to FIGS. 7 and 8. (The "undecided" transactions are
the transactions which are yet committed or aborted.) To enforce

5   the commitment order, transactions are selected for commitment
and transactions are selectively aborted by a commitment order
coordinator (COCO) 85, which is further described below with
reference to Figs. 10 and 15.

The present invention can be used in a single processing
10   system in which a plurality of transactions are performed
concurrently by scheduling component operations, or in a multi-
processor system in which component operations of the same
transaction are performed concurrently in different processors.
A multi-processor system 90 is illustrated in FIG. 4B. In this
15   case, three digital computers 91, 92, 93 are interconnected
through a communication channel 94, and the communication is
controlled by the transaction schedulers (TS) 95, 96, 97. In the
multi-processor system 90, any one of the transaction schedulers
95, 96, 97 may assume the role of a coordinator and issue global
20   transactions to the other transaction schedulers. These global
transactions are coordinated, for example, according to the well-
known two phase commit protocol, as was described above with
reference to the background art, and as further described below
with reference to FIGS. 14, 15 and 16.

25   The transaction schedulers may also exchange state
information over the communication channel 94. In particular,

transaction processing systems generally fall within two broad
categories called database management systems and object oriented
systems, depending upon whether or not state memory information
is resident in the non-volatile memory files of a particular one

5   of the digital computers 91, 92, 93, or whether the state
information is associated with predefined objects which may be
passed from one computer to another. The present invention,
however, is applicable to both types of systems because the
present invention more particularly concerns the scheduling of

10   component operations in the transactions and the enforcement of
commitment ordering, and is not particularly concerned with where
the state memory is physically located or maintained in a
distributed processing system.

       Turning now to FIG. 5, there is shown a flow chart of a

15   procedure followed by the transaction scheduler for real-time
scheduling of component operations of transactions in accordance
with available resources of the digital computer. In particular,
the transactions include input/output and memory access of
rotating memory such as disk drives, and possibly mathematical

20   computations that are performed by a coprocessor. Without real-
time scheduling, the central processing unit of the digital
computer would have to spend a good deal of time waiting for
these operations to be completed before performing the component
operations of other transactions.

25   To more effectively use the resources of the digital
computer, a transaction may dispatch input/output and memory

access requests to the input/output and memory units of the

computer, then set an inhibit flag indicating to the scheduler

that the processing of the current transaction should be

inhibited until completion of the input/output or memory access

5   operation, and finally execute a software interrupt to the

transaction scheduler in order to permit the transaction

scheduler to transfer execution to another transaction. When the

requested input/output or memory access operation is completed,

the input/output or memory device issues a completion interrupt

10   which is handled by a device handler interrupt routine that

clears the inhibit flag of the transaction that requested the

input/output or memory access operation. It should be noted that

input/output and memory access completion interrupts and device

handlers for such interrupts are well known in the art.

15   Referring now particularly to the first step 101 in FIG. 5,

the transaction scheduler responds to an interrupt by removing

the context of the interrupted transaction from the processor

stack of the digital computer, and by placing the context in a

respective context storage for the interrupted transaction. The

20   context includes the value of the program counter which points to

the interrupted memory location in the transaction program, as

well as the context of other general purpose registers in the

digital computer.

The transaction scheduler may also be entered during initial

25   start-up of the digital computer in step 102. In step 102, the

transaction list 83 and other data structures such as the

undecided serializability graph (USG) are cleared and pointers are initialized.

The transaction scheduler may also be entered at the end of preparation for a transaction. In this case, in step 103 the

5 transaction is marked to indicate that it is ready to be committed, and also the current time indicated by the real time clock (25 in FIG. 1) is saved in a memory location allocated to the transaction to indicate the time at which the transaction became ready. It should be noted, however, that some tasks

10 placed on the transaction list might be so-called background tasks of low priority, which are never completed and use central processor execution time remaining after the servicing of all transactions in the list.

The transaction scheduler may also be entered at the end of

15 a device handler interrupt routine. Step 111, for example, clears the inhibit flag (I in the list of FIG. 6) for the transaction having requested the input/output or memory operation, and then execution continues in step 101 to interrupt the current transaction to possibly reschedule execution back to

20 the transaction having requested the input/output or memory operation.

The transaction scheduler performs three major tasks; it responds to transaction requests by placing the transactions on the transaction list; it initiates the committing of ready

25 transactions, and it also schedules the performance of component operations of transactions. In step 104, for example, the

transaction scheduler checks whether a transaction has been requested. A transaction scheduler interrupt, for example, may occur in response to an interrupt signal from the input/output unit indicating that a user or another digital computer has

5   requested the performance of a transaction. In this case, in step 105 the transaction request is placed on the transaction list.

Turning for a moment to FIG. 6, there is shown a specific example of the transaction list 83. The transaction list

10   includes a linked list of transaction identification numbers 106. Associated with each transaction identification number is a pointer to the next entry in the linked list, and values for a number of flags (V, R, I, G, P). These flags include a valid flag V indicating whether the entry in the list includes valid

15   data, a flag R indicating whether preparation of the transaction has been completed and the transaction is ready to be committed, a flag I indicating whether preparation of the transaction has been inhibited until completion of an input/output or memory access request, a flag G indicating whether the transaction is a

20   local or global transaction, and a flag P indicating whether the completion of preparation of a global transaction has been reported to a coordinator. The flags G and P associated with global transactions are further described below with reference to FIGS. 14 and 15.

25   Also associated with the list 83 are a head pointer 108, a tail pointer 109, and a pointer 110 to the transaction being

performed. The head pointer 108, for example, has a negative

value when the list is empty, and otherwise has a positive value

pointing the list entry for the first (highest priority)

transaction. In a similar fashion, the tail pointer 109 has a

5   negative value when the list is empty and otherwise has a

positive value pointing to the last entry in the list. The

pointer 110 to the transaction being performed is used by the

transaction scheduler in step 101 of FIG. 5 when responding to an

interrupt. In particular the pointer 110 is used to find the

10   respective context storage location for the interrupted

transaction when performing step 101.

Returning now to FIG. 5, in step 112 the transaction

scheduler checks whether a transaction is ready to be committed.

If so, then in step 113 the transaction scheduler invokes the

15   commitment order coordinator (85) to select the transaction to

commit, and to enforce commitment ordering with possible aborts

and delay. When the commitment order coordinator decides not to

delay commitment, then in step 115 the resource manager (RM)

commits the results of the transaction to the state memory.

20   Finally, in step 116, the transaction scheduler checks the

transaction list to determine whether there is an uninhibited

transaction that is not yet ready. If so, then in step 117, the

transaction scheduler selects one of the uninhibited transactions

that is not yet ready. To perform 116 and 117, for example, the

25   transaction scheduler first checks whether the transaction list

is empty by testing whether the head pointer 108 has a negative

value. If the head pointer has a positive value, then the transaction scheduler checks the flags R and I for the transaction at the head of the list to determine whether is not yet ready and is not inhibited. If the first entry is ready or

5   is inhibited, then the transaction scheduler checks the tail pointer 109 to determine whether the end of the list has been reached. If not, then the transaction scheduler checks the pointer to the next entry and performs the same steps until either an uninhibited transaction not yet ready is found or the

10   end of the list has been reached.

When an uninhibited transaction not yet ready has been selected, then in step 118 the context of the selected transaction is placed on the stack. In this regard, it should be noted that when a transaction is first placed on the transaction

15   que, then an initial context for the transaction is placed in the respective context storage for the interrupted transaction. The initial context, for example, includes a program counter value pointing to the first instruction in the program for the transaction. After step 118, a return from interrupt is

20   performed in step 119 to begin or continue the execution of instructions in the program for the selected transaction.

Turning now to FIG. 7, there is shown a specific example of a data structure for storing the undecided transactions serializability graph (USG). Whenever a particular order of

25   performing conflicting operations in a respective pair of transactions has been established, that order of performance of

the conflicting operation is noted in the undecided transactions serializability graph. If the memory access operations performed by each transaction and the memory locations of those memory accessed operations are known at the time that a transaction is

5   placed on the list, then it is possible in Case 1 of FIG. 3A for the order of performing the conflicting operations to be determined at that time. In this regard, it should be noted that for Case 1 as illustrated in FIG. 3A, write operations are in effect performed at the time of transaction commitment. Aside

10   from this particular case, the order of performance of conflicting operations is determined when a second one of the conflicting operations is scheduled for performance by the transaction scheduler and the memory location accessed by that conflicting operation is determined.

15   At that time the presence of a conflict is detected, as further described below with reference to FIG. 11, the order of performance is recorded in the undecided transactions serializability graph. The data in the graph of FIG. 7 is presented in pictorial form in FIG. 8. The flags that are set in

20   the data structure of FIG. 7 correspond to edges 131 in the pictorial representation of FIG. 8. The direction of an edge 131 indicates the order of performance of the conflicting operations in the transactions. Once this order of performance is established, it is enforced by aborting transactions or delaying

25   transactions or additional conflicting operations.

Enforcement of the commitment order by aborting transactions
is illustrated by steps 141 and 142 in FIG. 9. In step 141 a
ready transaction to be committed is selected. In situations
where there are a number of ready transactions, a particular

5   ready transaction is selected, for example, by selecting the
first such transaction in the list, by comparing predetermined
priorities associated with the transactions, or in response to
the selection from a coordinator. For the list 106 in FIG. 6,
for example, global transactions could be given priority over

10   local transactions in the selection process.

In step 142 the commitment order is enforced by aborting the
transactions for which commitment is contrary to the commitment
order and commitment of the selected transaction. For the
commitment order illustrated by the graph in FIG. 8, for example,

15   if the transaction T_1 is selected, then transactions T_0 and T_3 are
aborted to enforce the commitment order. Aborting of a
transaction involves discarding the results of any transaction.
For local transactions, a transaction could be aborted by
resetting the contents of its respective context storage to its

20   initial context. In other words, the current value of the
program counter for the transaction is reset to the beginning of
the program for the transaction. In addition, the transaction
list 106 and the undecided transactions serializability graph 84
must be reinitialized. For global transactions, an aborted

25   transaction is restarted if at all by the coordinator. In this

case, the transaction is entirely removed from the transaction list.

Turning now to FIG. 10, there is shown a flow chart generally designated 150 of a procedure for selecting a ready

5   transaction to be committed, as introduced above with respect to step 141 of FIG. 9. In the first step 151 of FIG. 10, the transaction list 106 is searched beginning at the head of the list for the first ready transaction. In addition, a "commit flag" used later in FIG. 10 is cleared. Next in step 152 the

10   number of members in the selected transaction's "abort set" is counted. With reference to the undecided transaction serializability graph 84 in FIG. 8, for example, the abort set of each transaction in the graph includes its preceding transactions. In other words, for the transaction T_1, its abort

15   set includes transactions T_0 and T_3, and the number and members in T_1's abort set is two. In a similar fashion, the transactions T_0 and T_4 have no members in their abort set. With reference to the specific data structure in FIG. 7, the members of a transaction's abort set is determined by the edge flags that are

20   set along a column of the data structure, and the number of members in the abort set is computed by counting the number of flags that are set in the respective column for the transaction. If the selected transaction's abort set is null (has no members), as checked in step 153, then in step 154 execution continues to

25   possibly commit the results of the transaction to state memory, as will be further described below with reference to FIG. 15 for

a transaction processing system handling both local and global transactions.

When the abort set of the selected transaction is not null, an attempt is made to delay the committing of the transaction so that the number of members in its abort set may decrease. In this regard it should be noted that the abort set of a ready transaction can never increase, but it may decrease, since it will decrease by one each time that a member of its abort set is committed. The committing of a selected transaction, however, should not be delayed indefinitely, or otherwise the system may become blocked. Therefore the delaying of a selected transaction will be terminated when the delay exceeds a predetermined delay time. For processing global transactions, it is also desirable to terminate the delay in response to a termination signal from the coordinator, as further described below with reference to FIG. 15. In step 157 of FIG. 10, the delay time is computed as the difference between the current time and the ready time for the transaction (which was saved in step 103 of FIG. 5). When the delay time exceeds a predetermined limit, as checked in step 158, then in step 159 a commit flag is set.

In step 160, when there are additional ready transactions in the transaction list, execution branches to step 156 to get the next ready transaction, and execution loops back to step 152. Otherwise, in step 161, the commit flag is inspected, and if it is not set then execution continues without a decision having been made to commit any of the ready transactions. Otherwise, in

step 162 the ready transaction with the minimum number of members in its abort set is selected.

Turning now to FIG. 11, there is shown a flow chart 170 of a procedure for determining the commitment order of conflicting

5   component operations of transactions. The procedure 170 is invoked during the preparation of a memory access operation such as a read or write. In the first step 171, the address of the memory access operation is determined. Next, in step 172 the address is compared to addresses of prior operations that may

10   conflict. This is done by searching a list of addresses of prior operations for each transaction in the transaction list. If the present operation is a read operation, then the read operation may conflict with prior write operations. If the present operation is a write operation, then the write operation may

15   conflict with a prior read (or for Case 2 of FIG. 3B, a prior write operation). When there is an address match as tested in step 173, then in step 180 the present order of the transaction is recorded the undecided transactions serializability graph (84 in FIG. 7). In particular, for Case 1 of FIG. 3A, conflicts only

20   occur between a read operation and a write operation, and the order of operation is read then write. For Case 2 of FIG. 3B, the present order must be for the current transaction to be performed after the previous transaction. In step 181 execution branches back to step 172 if there are additional prior memory

25   access operations to check; or otherwise preparation of the memory access continues in step 177 by adding the address

determined in step 171 to a list of addresses for read or write

operations of the current transaction. Then, in step 178, the

operation is prepared or performed in step 178. Execution then

returns to the transaction.

5   Turning now to FIG. 12, there is shown an augmented

undecided transaction serializability graph in which edges for

write read conflicts are distinguished from edges of other

conflicts. Such an augmented graph can be stored in a data

structure similar to the data structure shown in FIG. 7, but each

10   edge is represented by a pair of flags, consisting of a first

flag indicating any kind of conflict, and a second flag

indicating a write-read conflict. The augmented graph of FIG. 12

is used to perform cascading aborts to insure recoverability for

a system in which a second transaction can read the write data of

15   a first transaction before the first transaction is committed, as

was described above with reference to FIG. 3B. A procedure for

performing a cascading abort is shown in the flow chart 190 of

FIG. 13. Suppose, for example, that T_5 is selected as a ready

transaction to be committed. To enforce commitment ordering,

20   then transactions T_3 and T_4 of FIG. 12 must be aborted. However,

assume that the transaction processing system operates in the

fashion as described above with reference to FIG. 3B. In this

case, when a transaction is aborted to enforce commitment

ordering, then every transaction that has read write data of the

25   aborted transaction must also be aborted. From the augmented

graph of FIG. 12, it is seen that when the transaction T_4 is

aborted, then the transaction T_7 must also be aborted because the transaction T_7 because of the write read conflict between transactions T_4 and T_7. Moreover, when the transaction T_7 is aborted, then so must the transaction T_8 because the transaction

5   T_8 has read data written by the transaction T_7.

Returning now to FIG. 13, in the cascading abort procedure 190 for aborting the transaction T_x, in the first step 191 the augmented graph is searched to find all of the transactions T_y such that T_y has read data written by T_x. Then in step 192 the

10   transaction T_x is aborted. In a final step 192, the subroutine 190 of FIG. 13 is recursively called to abort each of the transactions T_y.

Turning now to FIG. 14, there is shown a state diagram of a transaction processing system that processes both local and

15   global transactions. The local transactions, for example, are issued by a local user 201, and the global transactions are issued by a coordinator 202. In each case the transaction scheduler receives the transaction request and puts the transaction request into an entry of the transaction list. At

20   this point, the transaction is said to be in preparation. The transaction scheduler eventually transfers execution to the transaction in preparation, and the transaction is executed until either it becomes inhibited or it becomes ready. As described above in connection with FIG. 5, a transaction may inhibit itself

25   after requesting an input/output or memory operation, an upon completion of the input/output or memory operation, the

transaction will become uninhibited. A transaction that is either in preparation, inhibited or ready can be aborted to enforce commitment ordering.

The transaction scheduler may commit a ready local

5   transaction. To insure global synchronization in a distributed transaction processing system, however, a ready global transaction is committed only after a handshake with the coordinator. This handshake insures that a global transaction is not committed unless all of the processors that are processing

10 assigned portions of the transaction are also ready to commit their assigned portions of the global transaction. Therefore, when the transaction scheduler changes the state of a global transaction from the "in preparation" to the "ready" state, the transaction scheduler transmits a "prepared" signal to the

15 coordinator 202.

When the coordinator 202 receives "prepared" signals from all of the transaction schedulers participating in a transaction, then the coordinator sends a "commit" command back to the transaction scheduler. If, however, the coordinator fails to

20 receive a "prepared" signal from all of the participating transaction schedulers, then the coordinator may transmit an "abort" signal to the transaction scheduler. In FIG. 14, these handshake signals are indicated by dotted lines.

When a local transaction is committed, the transaction

25 scheduler removes the transaction from the list and notifies the local user that the transaction has been completed. In a similar

fashion, when a global transaction is committed, the transaction scheduler removes the global from the transaction list and sends a signal to the coordinator indicating that the transaction has been committed. Moreover, when a global transaction is aborted,

5   the global transaction is removed from the transaction list and the undecided transaction serializability graph, and the transaction scheduler sends a signal to the coordinator to confirm the abort. For a local transaction, however, it may be desirable to restart preparation of the transaction, and in this

10   case it is only necessary to reset the initial context of the transaction, clear the undecided transactions serializability graph, and set the state of the transaction back to "in preparation" by resetting R and I flags in the list entry of the transaction.

15   Turning now to FIG. 15, there is shown a flow chart of the steps followed by the commitment coordinator for scheduling the commitment of global transactions. The procedure in FIG. 10, for example, may have decided to continue without committing a transaction. In step 211, however, the transaction scheduler may

20   have received a "terminate delay signal" from the coordinator. This may be a distinct signal from the coordinator, or it could be a retransmission of a previously transmitted transaction request. When such a signal is received, then in step 212 the R flag for the transaction in the transaction list is inspected to

25   determine whether the global transaction is ready. If not, then the global transaction cannot be committed. Otherwise, then in

step 213 the members of the abort set for the transaction are
inspected to determine whether any of the members is a global
transaction with the P flag set. Step 213 is also performed with
global or local transactions for which the procedure in FIG. 10

5   selected the transaction to be committed if possible. If the
abort set for the transaction selected to be committed includes a
global transaction with the P flag set, then the selected
transaction cannot be committed before the global transaction
with P flag set. Therefore, execution continues in the

10   transaction scheduler. Otherwise, in step 214 execution branches
depending upon whether the transaction to be committed is a
global transaction. If it is a global transaction, it cannot be
committed until after the transaction scheduler sends a prepare
acknowledge signal to the coordinator, and receives back a

15   confirming "commit" signal. Therefore, for a global transaction,
in step 216 the prepare acknowledge signal ("PREPARED" in FIG.
14) is sent to the coordinator, and the P flag for the global
transaction in the transaction list is set, and execution
continues in the transaction scheduler. Otherwise, for a local

20   transaction to be committed, in step 215 the transactions in the
abort set are aborted (and the "cascading" abort of FIG. 13 is
used for Case 2 of FIG. 3A), and the results of the selected
local transaction are committed to state memory. Execution then
continues in the transaction scheduler.

25   Turning now to FIG. 16, there is shown a flow chart 220 for
handling prepared global transactions in response to a commit or

an abort interrupt from the coordinator. In the first step 221, execution branches depending upon whether the interrupt is for an abort or a commit. In the case of an abort, in step 222 the global transaction is aborted by removing the transaction from

5   the transaction list and clearing corresponding flags in the undecided transactions serializability graph. Finally, in step 223, an acknowledgement of the abort is sent back to the coordinator and then execution returns from the interrupt.

If it was determined in step 221 that the interrupt was for

10   a commit, then in step 224 the transactions in the abort set of the selected global transaction are aborted, and the results of the selected transaction are committed to state memory. Then in step 223 an acknowledgement of the commitment is sent back to the coordinator and execution returns from the interrupt.

15   Turning now to FIG. 17, there is shown an embodiment of the present invention wherein a commitment order coordinator (COCO) 251 is inserted into a conventional transaction processing system having a transaction manager (TM) 252 and a resorce manager (RM) 253. As shown, the commitment order coordinator 251 assumes a

20   subset of the interface 254 between the transaction manager 252 and the resource manager 253. The commitment order coordinator 251 intercepts a conventional portion 255 of the interface 254, and is interconnected to the resource manager 253 via an extended interface 256 that has some additional signals unique to the

25   operation of the commitment order coordinator. In a distributed

processing system, the configuration shown in FIG. 17 may be used at each node in the system.

It is assumed that the resource manager 253 provides the following conventional services:

5   R_PREPARE(T): The TM notifies the RM to complete the transaction T. It means that the RM will not receive any additional requests or external data on behalf of transaction T;

R_COMMIT(T): The TM notifies the RM to commit transaction T. A prerequisite for invocation of this service is that the RM

10   has previously acknowledged preparation of the transaction (i.e., voted YES); and

R_ABORT(T): The TM notifies the RM (and eventually also all the other RMs involved with T) to abort transaction T.

It is also assumed that the transaction manager 252 provides

15   the following conventional services:

T_READY(T): The RM notifies the TM that is has completed processing transaction T, and it votes YES (i.e. it is ready to commit or abort T according to the TM's notification);

T_ABORT(T): The RM notifies the TM that it has aborted

20   transaction T (which will result in aborting T by all the RMs involved).

When inserted into the TM-RM interface 254, the commitment order coordinator 251, rather than the resource manager 253, directly invokes T_READY(T) and T_ABORT(T) services of the

25   transaction manager 252. Moreover, the commitment order coordinator 251, instead of the resource manager directly

R_ABORT(T) services. The resource manager 253 is also invoked by the commitment order coordinator 251 to provide the following is additional service: R_CONFLICT_ACK(T_1, T_2). After this invocation the RM can execute the operation in T_2 that causes the

5 respective conflict with T_1.

Keeping in mind the above definitions of the invocations in the system of FIG. 17, it should be apparent that states of the system are responsive to the invocations as shown in FIG. 18. The invocations are further defined by the following PASCAL/SQL

10 based pseudo code, which is based on the state diagram of FIG. 18.

In the following pseudo code, concurrent invocations are allowed, as well as several concurrent invocations of the same service. T_ERROR(T) and R_ERROR(T) are error message invocations

15 of the TM, RM respectively, indicating erroneous invocation sequencing. The transaction's atomic state is returned as an error type.

```
C_T_COMMIT(T)
```

20
```
if not C_ready(T)  then T_ERROR(T)

else
```

25 begin
```
        set committed (T)                        /* state change */

        for every T' in ABORT(T)
```
30
```
                if not aborted (T') then

                        begin

                                T_ABORT(T')          /* abort T' */
```
35

```
                        set C_aborted (T')   /* state change for t" */

                        end

5
        R_COMMIT(T)

        delete T from the USG                /* set deleted (T);

10                                            pseudo state change */

    end

    end_C_T_COMMIT

15


    C_T_ABORT(T)

20
    if not (undecided (T) or C_aborted(T)) then T_ERROR(T)

    else

25  begin    /* set T_aborted(T)      pseudo state change */

             R_ABORT(T)               /* notify the RM to abort T */

             delete T from the USG    /* set deleted(T);

30                                        pseudo state change */

    end

35  end_C_T_ABORT



40  C_R_BEGIN(T)

    if not non_existent(T)

    /* i.e. if there is a T node in the USG*/) then R_ERROR(T)

45  else

    begin

50           insert a node for T to the USG

             set wait(T)                     /* state change */
```

```
        end

        end_C_R_BEGIN

 5



10      C_R_READY(T)

        if not wait(T) then R_ERROR(T)

        else
15
        begin

                set R_ready(T)                    /* state change */

20              set time(T):=local_time

                    /* time(T) is used by the VOTE procedure to timeout
                       delaying the voting on T; local_time indicates the loca
                       clock's value */
25
        end

        enc_C_R_READY

30


        C_R_ABORT(T)

35 ···  if not(active(T) or   /*in a case of invocation collision */

        C_aborted(T) or deleted(T)      /* i.e. T does not exist */)

        then R_ERROR(T)
40
        else if active(T) then

                being

45                          /* set T_aborted(T)      pseudo state change */

                            T_ABORT(T)

                            set C_aborted(T)         /* state change */
50
                end
```

```
        end_C_R_ABORT


5

        C_R_CONFLICT(T',T)

        if not wait(T) or deleted (T') then R_ERROR(T)
10
        else

        begin
15              /* set conflict_insert(T)      pseudo state change */

                if the edge (T',T) does not exist in the USG then

                        insert the edge (T',T) to the USG
20
                R_CONFLICT_ACK(T',T)

                /* set wait(T)                          pseudo change */

25      end

        end_C_R_CONFLICT


30

        VOTE (timeout)

        /* timeout:  A YES vote may be issued on a ready transaction T
35      after the time passed since T entered a ready state has exceeded
        the timeout value, even if this may cause aborting transactions
        in the set ABORT(T). */

        repeat
40
                select any transaction T where

                                R_ready(T)

45                      and     (ABORT(T) is empty

                                or   decided(T') for every T' in ABORT(T))

                if found then YES(T)      /* voting YES without aborts */
50
                else        begin
```

```
              select any transaction T where
                    R_ready(T)
                    and not C_ready(T') for every transaction T'
                                                     in ABORT(T)

                            /* YES vote on T' has not been invoked
                    and optimal(T)    /*(i.e., T is chosen to
                            minimize the cost of aborting
                            transactions in T's abort set)*/
                    and local_time - time(T) > timeout
                            /* T entered the ready state at time(T);
                            local_time indicates the clock's value
              if found then YES(T)    /*   voting YES with
                                            possible aborts */

              end
end_repeat
YES (T)

                            /* procedure */
        T_READY(T)          /* voting YES on T */
        set C_ready(T)      /* state change */
end_YES
end_VOTE
```

The commitment order coordinator 251 as defined by the above pseudo code can be modified to guarantee recoverability of the output schedule. The modified commitment coordinator will be referred to as a CORCO. The CORCO is used when the interfacing

RM (253 in FIG. 17) does not guarantee recoverability, and it still follows the state diagram of FIG. 18. The CORCO differs from the above pseudo code example by using cascading aborts, and by the following additional modifications. Write-read conflicts

5   are reflected in the USG's edges, as illustrated in FIG. 12. If the edge (T_1, T_2) represents a $\underline{wr}$ conflict (and possibly some other conflicts) the Boolean $\underline{wr}(T_1, T)$ has the value $\underline{true}$, and no YES vote is issued on T_2 if $\underline{wr}(T_1, T_2)$ has the value $\underline{true}$ (to avoid recoverability violation). Moreover, the service C_R_CONFLICT

10 has an additional Boolean parameter, $\underline{wr}$, to indicate a $\underline{wr}$ conflict (C_R_CONFLICT$(T_1, T_2, \underline{wr})$). Furthermore, the CORCO's invocations, as well as the its VOTE procedure, are modification of the COCO's, reflecting the differences listed above.

The following recursive procedure CASCADE(T) invokes

15 T_ABORT(T) and generates additional T_ABORT invocations when necessary to maintain recoverability.

CASCADE(T)   /* recursive procedure; invokes T_ABORT for T and all related transactions to be aborted to maintain Recoverability

20 (cascading aborts) */

T_ABORT(T)

set C_aborted(T)   /* state change */

for every T' such that the edge (T, T') is in the USG

           if $\underline{wr}(T, T')$ then CASCADE(T')

25 end_CASCADE

```
     C_T_COMMIT(T)

     if not C_ready(T) then T_ERROR(T)

 5   else

     begin           set committed(T)                /* state change */

                     for every T' in ABORT(T)
10
                             if no aborted(T') then CASCADE(T')

                     R_COMMIT(T)

15                   delete T from the USG           /* set deleted(T);

                                                     pseudo state change */

     end
20
     end_C_T_COMMIT


     C_T_ABORT(T)
25
     if not( undecided(T) or C_aborted(T) then T_ERROR(R)

     else

30   begin     /* set T_aborted(T)        pseudo state change */

               for every T' such that the edge (T,T') is in the USG

                       if wr(T,T') then CASCADE(T')
35
               R_ABORT(T)                    /* notify the RM to abort T */

               delete T from the USG    /* set deleted(T);

40                                          pseudo state change */

     end

     enc_C_T_ABORT
45
       C_R_BEGIN(T)                          /* unchanged; same as the COCO's *,

50   C_R_READY(T)                            /* unchanged; same as the COCO's *,
```

```
     C_R_ABORT(T)

     if not( active(T) or   /* in a case of invocation collision */
5    C_aborted(T) or deleted(T)      /* i.e. T does not exist */  )

     then R_ERROR(T)

     else if active(T) then
10
                    - begin

                        /* set T_aborted(T)      pseudo state change */
15                      CASCADE(T);

             end

     end_C_R_ABORT
20

     C_R_CONFLICT(T',T,wr)

     if not wait(T) or deleted(T') then R_ERROR(T)
25
     else       begin    /* set conflict_insert(T)  pseudo state change */
                         if the edge (T',T) does not exist in the USG then
30                              begin  insert the edge (T',T) := false
                              if wr then set wr(T',T) := true
                              R_CONFLICT_ACK(T',T,wr)
35                            /* set wait(T)              pseudo change */
             end
40   end_C_R_CONFLICT
```

/* Note: The parameter wr in R_CONFLICT_ACK is used since the
45   RM does not match a conflict with its ack, and multiple conflicts
between the same two transactions may be invoked concurrently

(one such ack guarantees a proper edge in the USG; once wr=true, the edge type is set, and remains unchanged). */

```
VOTE (timeout)

5      /* timeout:  A YES vote may be issued on a ready transaction T
       after the time passed since T entered a ready state has exceeded
       the timeout value, even if this may cause aborting transactions
       in the set ABORT(T). */
       repeat

10           select any transaction T where
                        R_ready(T)
             and         (ABORT(T) is empty
                          or (aborted(T') and not wr(T',T) or committed(T')]
                                    for every T' in ABORT(T)   )
15           if found then YES(T)          /* voting YES without aborts */
             else     begin
                          select any transaction T where
                                    R_ready(T)
                          and
20                                (not C_ready(T')
                                    /* YES vote on T' has not been invoked
                                    and not wr(T',T)  ).
                                    for every transaction T' in ABORT(T)
                          and     optimal(T)
25                        and     local_time - time(T) > timeout
                                    /* T entered the ready state at time(T);
```

```
                                       local_time indicates the clock's value *.
                     if found then YES(T)  /* voting YES with

                                                      possible aborts */
                end
5    end_repeat


     YES(T)

                                                /* procedure*/
10                   T_READY(T)                 /* voting YES on T */
                     set C_ready(T)             /* state change */
     end_YES
     end_VOTE
```

15   In view of the above, the component operations of a number
of transactions can be distributed and scheduled for execution in
a manner most efficient for use of available computing
capabilities, yet consistency can be maintained by enforcing a
commitment ordering in which transactions are committed in the
20   same order as the order of performance of conflicting operations.
In a distributed transaction processing system, conformance to
commitment ordering guarantees the serializability of the
combined (global) schedule. Moreover, the serializability of the
combined (global) schedule is maintained while maintaining the
25   autonomy of each distributed processor, and while using a

conventional atomic commitment protocol for coordination of global transactions.

APPENDIX I

DEFINITIONS AND PROOFS OF COMMITMENT ORDERING PROPERTIES

5   The following shows that a history property called
"Commitment Ordering" solves the global serializability problem.
In particular, global serializability is guaranteed if each
resource manager in a distributed transaction processing system
follows "commitment ordering," and if the resource managers are
10   "autonomous" (i.e., they coordinate via atomic commitment
protocols only, and do not exchange any additional concurrency
control information.)

Definitions:

15   1.   A <u>transaction</u> T_i is a <u>partial order</u> of events. The binary,
asymmetric, transitive, and irreflexive relation that comprises
the partial order is denoted "$<_i$". The subscript i may be
omitted when the transaction's identifier is known from the
context. Events include read and write operations; $r_i[x]$ denotes
20   that transaction T_i has read data item x, and $w_i[x]$ means that
transaction T_i has written data item x. A transaction has also
an event of ending a transaction; e_i means that T_i has ended.

Axioms:

25   2.   A transaction T_i has exactly a single event e_i. A value is
assigned to e_i as follows: $e_i = c$ if and only if the transaction

is committed; $e_i = a$ if and only if the transaction is aborted. e_i may be denoted c_i or a_i when $e_i = c$ or $e_i = a$ respectively.

3.   For any operation $p_i[x]$ that is either $r_i[x]$ or $w_i[x]$, $p_i[x]$

5    $<_i e_i$.

Definitions:

4.   Two operations on a data item x, $p_i[x]$, $q_j[x]$ are conflicting if either $p_i[x]$ is $w_i[x]$ or $q_j[x]$ is $w_j[x]$.

10

5.   A complete history H over a set T of transactions is a partial order with a relation $<_H$ defined according to the following axioms 6, 7 and 8.

15   Axioms:

6.   If T_i is in T and $event_a <_i event_b$ then $event_a <_H event_b$.

7.   If T_i and T_j are in T then for any two conflicting operations $p_i[x]$, $q_j[x]$, either $p_i[x] <_H q_j[x]$ or $q_j[x] <_H p_i[x]$.

20

8.   Let T_i, T_j be transactions in T and $q_j[x]$ any operation. If $w_i[x] <_H q_j[x]$ then either $e_i <_H q_j[x]$ or $q_j[x] <_H e_i$. (This axiom provides a unique definition of the history's semantics because if $e_i = a$ the effect of $w_i[x]$ is undone; i.e. reading x after e_i

25   results in reading a value of x that existed just before $w_i[x]$.)

(Note: The subscript H in $<_H$ may be omitted when H is known from the context.)

Definitions:

5   9. A <u>history</u> is any <u>prefix</u> of a complete history. A <u>prefix</u> of a partial order P over a set S is a partial order P' over a set S'$\subseteq$ S, with the following properties:

If b $\in$ S' and a $<_P$b then also a $\in$ S'

10   If a,b $\in$ S' then a $<_P$b if and only if a $<_{P'}$b

10. A transaction T_2 <u>is in conflict with</u> transaction T_1 if and only if for respective conflicting operations $q_2[x]$, $p_1[x]$, $p_1[x] < q_2[x]$. (Note that this definition is asymmetric.)

15

11. If $p_1[x]$ is $w_1[x]$ and $q_2[x]$ is $w_2[x]$ then T_2 <u>is in a ww conflict with</u> transaction T_1.

12. If $p_1[x]$ is $w_1[x]$ and $q_2[x]$ is $r_2[x]$ then T_2 <u>is in a wr</u>

20 <u>conflict with</u> transaction T_1.

13. If p_1 is $r_1[x]$ and $q_2[x]$ is $w_2[x]$ then T_2 <u>is in a rw conflict with</u> transaction T_1.

25 14. There is a <u>conflict equivalence</u> between two histories H and H' (the two are <u>conflict equivalent</u>) if and only if they are both

defined over the same set of transactions T, and consist of the same transaction events (for partially executed transactions), and $p_i[x] <_H q_j[x]$ if and only if $p_i[x] <_{H'} q_j[x]$ for any conflicting operations $p_i[x]$, $q_j[x]$ of any committed transaction T_i, T_j respectively in T (i.e. H and H' have the same conflicts between operations of committed transactions).

15. A history H over a transaction set T is serial if and only if for every two transaction T_i, T_j in T the following is true: If $p_i[x] <_H q_j[y]$ then for any other operations $s_i[u]$, $t_j[v]$ in H $s_i[u] <_H t_j[v]$ (i.e. all the operations of T_i precede all the operations of T_j).

16. A history is serializable (SER; is in SER) if and only if it is conflict equivalent to some serial history.

17. A Serializability Graph of a history H, SG(H), is the directed graph SG(H) = (T,C) wherein T is the set of all unaborted (i.e. committed and incomplete) transaction in H, and C (a subset of TxT) is a set of edges that represent transaction conflicts such that for any two transactions T_1, T_2 in T, there is an edge from T_1 to T_2 if and only if T_2 is in a conflict with T_1. The Committed Transactions Serializability Graph of a history H, CSG(H), is the subgraph of SG(H) with all the committed transactions as nodes and with all respective edges. The Undecided Transactions Serializability Graph of a history H,

USG(H), is the subgraph of SG(H) with all the uncommitted (i.e., incomplete) transactions as nodes and with all the respective edges.

5   The Serializability theorem:

18. A history H is _serializable_ is (SER) if and only if CSG(H) is cycle-free.

Definitions:

10 19. A history H is _recoverable_ (REC; is in REC) if and only if for any two transactions T_1, T_2 in H, whenever T_2 is in a _wr_ conflict with T_1, T_2 is committed only after T_1 has been committed. Formally: ($w_1[x] < r_2[x]$ and $e_2 = c$) implies (($e_1 < e_2$ and $e_1 = c$) or ($e_1 < r_2[x]$ and $e_1 = a$)).

15

20. A history H _avoids cascading aborts_ (ACA; is in ACA) if and only if any transaction in H reads data written by committed transactions only. Let T_1, T_2 be any two transactions in H. The following expression is a formal representation of this concept:

20      $w_1[x] < r_2[x]$ implies $e_1 < r_2[x]$.

21. Let T_1, T_2 be any two transactions in H. H is _strict_ (ST; is in ST; has the _strictness_ property) if and only if $w_1[x] < p_2[x]$ implies $e_1 < p_2[x]$ where $p_2[x]$ is either $r_2[x]$ or $w_2[x]$.

25

Theorem:

22.   REC $\supset$ ACA $\supset$ ST where $\supset$ denotes a strict containment. (This theorem follows immediately from the definitions.)

Definitions:

5   23. Two Phase Locking (2PL) is a serializability mechanism that implements two types of locks: write locks and read locks. It consists of partitioning a transaction's duration of two phases: In the first locks are acquired; in the second locks are released.

10

24. A history is in Strong-Strict Two-phase Locking (S-S2PL) if and only if for any conflicting operations $p_1[x]$, $q_2[x]$ of transactions T_1, T_2 respectively in H $p_1[x] < q_2[x]$ implies $e_1 < q_2[x]$. (Note: A history is a two-phase locking if it can be

15 generated by the two-phase locking mechanism. Strict two-phase locking requires that write locks issued on behalf of a transaction are not released until its end; however, read locks can be released earlier, at the end of phase one of the two-phase locking mechanism. Strong-Strict Two-Phase locking requires that

20 all locks are not released before the transaction ends (when it is either committed or aborted.) Strong-Strict Two-Phase locking blocks any conflicting operations on a data item accessed by a transaction until the end of the transaction.)

30. A transaction is <u>undecided</u> if and only if it is either <u>ready</u> or <u>active</u>.

Commitment Ordering Definition:

5   31. A history has the <u>Commitment Ordering</u> property (i.e., is in CO) if and only if for any conflicting operations $p_1[x]$, $q_2[x]$ of committed transactions T_1, T_2 respectively, $p_1[x] < q_2[x]$ implies $e_1 < e_2$. Formally: $(e_1 = c$ and $e_2 = c$ and $p_1[x] < q_2[x])$ implies $e_1 < e_2$.

10

Commitment Ordering Theorem:

32. SER $\supset$ CO (i.e., Commitment Ordering Implies Serializability.)

Proof: Let a history H be a CO, and let $... \rightarrow T_i \rightarrow ... \rightarrow$
15   $T_j \rightarrow ...$ be a (directed) path in CSG(H). Using the CO definition and induction θ on the order on the path we conclude immediately that $c_i < c_j$. Now suppose that H is not in SER. By the Serializability Theorem (1.18) (without loss of generality) there is a cycle $T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_n \rightarrow T_1$ in CSG(H). First let T_i
20   and T_j above be T_1 and T_2 respectively (consider an appropriate prefix of the path above). This implies by the observation above that $c_1 < c_2$. Now let T_i and T_j be T_2 and T_1 respectively (consider an appropriate suffix of the path above). This implies that $c_2 < c_1$. However, the two implications contradict since the
25   relation "<" is asymmetric. Hence CSG(H) is acyclic, and H is in SER by the Serializability Theorem. Now examine the following

serializable, non CO history to conclude that the containment is
strict: $w_1[x]$ $w_2[x]$ c_2 c_1

Definitions:

5   33. <u>Timestamp Ordering</u> (TO) concurrency control mechanisms
provide serializability and are based on a time-stamp $ts(T_i)$ (a
real number) associated with each transaction T_i; timestamps are
distinct.

10   Timestamp Ordering Rule:
34. For any two conflicting operations $p_1[x]$, $q_2[x]$ of any
<u>committed</u> transactions T_1, T_2 respectively, $ts(T_1) < ts(T_2)$
implies $p_1[x] < q_2[x]$. (Note: timestamp ordering is non-
blocking (because it can be enforced by aborting either T_1 or T_2
15   <u>after</u> all their operations have been issued), and provides the
basis for optimistic timestamp ordering based concurrency
control, as well as a basis for blocking Timestamp Ordering based
mechanisms.

20   Blocking Timestamp Ordering Rule:
35. For any two conflicting operations $p_1[x]$, $q_2[x]$ of <u>any</u>
transactions T_1, T_2 respectively, $ts(T_1) < ts(T_2)$ implies $p_1[x] <
q_2[x]$. (Note: This Blocking Timestamp Ordering rule requires
that conflicting operations are scheduled according to the
25   timestamps order regardless of whether the transaction is
committed.)

Timestamp Commitment Ordering Rule:

36. For any two <u>committed</u> transaction T_1, T_2 with respective

conflicting operations, $ts(T_1) < ts(T_2)$ implies $e_1 < e_2$.

Formally: $(e_1=c$ and $e_2=c$ and $(p_1[x]$, $q_2[x]$ conflict) and

5    $ts(T_1) < ts(T_2))$ implies $e_1 < e_2$.

Theorem:

37. A history has the Commitment Ordering property if and only

if it is generated by a mechanism that obeys both the Timestamp

10   Ordering Rule (34) and the timestamp Commitment Ordering Rule

(36). (Note: This theorem means that if the Timestamp

Commitment Ordering (TCO) rule is being enforced by any Timestamp

Ordering mechanism, then only histories having the Commitment

Ordering Property are generated. The TCO rule can be easily

15   enforced by delaying commitment events when necessary to comply

with the timestamp order.)

Definitions:

38. <u>Transaction Termination Scheduler</u> (TTS) is a component that

20   monitors the set of ready transactions and decides when and which

transaction to commit or abort. In a multi-resource manager

environment this component participates in <u>atomic commitment</u>

procedures on behalf of its resource manager and controls (within

the respective resource manager) the execution of the decision

25   reached via atomic commitment for each relevant transaction.

39.   A Commitment Ordering Transaction Terminating Scheduler
(COTTS) performs the following procedure or its equivalent:

(a) The COTTS maintains a serializability graph, USC, of
all <u>undecided</u> transactions. Every new transaction processed by

5   the RM is reflected as a new node in USG; every conflict between
transactions in USG if reflected by a directed edge (an edge
between two transactions may represent several conflicts though).
USG(H) = (UT,C) where UT is the set of all undecided transactions
in a history H; and C (a subset of UTxUT) is the set of directed

10   edges between transactions in UT. There is an edge from T_1 to T_2
if and only if T_2 is in a conflict with T_1. The USG reflects all
operation's conflicts until commit. The set of transactions
aborted as a result of committing T (to prevent future
commitment-ordering violation) is defined as follows:

15   $ABORT_{co}(T) = \{T' \mid T' \rightarrow T$ is in $C\}$
The COTTS iteratively performs the following steps:
(a) selects any ready transaction (i.e. that has completed
processing) T in USG (using any criteria, possibly by priorities
assigned to each transaction; a priority can be changed

20   dynamically as long as the transaction is in USG), and commit it;

(b) aborts all the transactions in the set $ABORT_{co}(T)$, i.e.
all the transactions (both <u>ready</u> and <u>active</u>) in USG that have an
edge going to T; and (c) removes any <u>decided</u> transaction (T and
the aborted transactions) from the graph (they do not belong in

25   USG by definition).

Theorem:

40. The Commitment Ordering Transaction Termination Scheduler (COTTS) produces histories having the Commitment Ordering (CO) Property.

5   Proof: The proof is by induction on the number of iterations by the COTTS, starting from an empty history H_0, and an empty graph $USG_0 = USG(H_0)$. H_0 is CO. Assume that the history H_n, generated after iteration nb, is CO. USG_n (in its UT component) includes all the undecided transactions in H_n. Now

10   perform an additional iteration, number n+1, and commit transaction T_1 (without loss of generality - wlg) in USG_n. H_{n+1} includes all the transactions in H_n and the new (undecided) transactions that have been generated after completing step n (and are in USG_{n+1}). Examine the following cases after completing

15   iteration n+1:

(a) Let T_2, T_3 (wlg) be two committed transactions in H_n. If T_3 is in conflict with T_2 then $c_2 < c_3$ since H_n is CO by the induction hypothesis.

(b) $c_2 < c_1$ for every (previously) committed transaction T_2

20   in H_n with which T_1 is in a conflict.

The cases above exhaust all possible pairs of conflicting committed transactions in Hn1. Hence Hn+1 is CO. (Note: If there exists a transaction that does not reside on any cycle in USG then there exists a transaction T with no edges from any

25   other transaction. T can be committed without aborting any other transaction since $ABORT_{co}(T)$ is empty. If all the transactions

in USG are on cycles, at least one transaction has to be aborted. This situation seems to be uncommon. In a multi RM environment, if the RM (TTS) does not choose by itself the transaction to be committed, but rather receives a request (via an atomic

5   commitment protocol) to commit some transaction T in USG, all transactions in $ABORT_{co}(T)$, i.e. with edges to T, need to be aborted when T is committed (by the COTTS). The TTS may choose to commit T immediately (the non-blocking without delays approach). Another approach (non-blocking with delays) is to

10   delay the commitment for a given amount of time. During the delay the set $ABORT_{co}(T)$ may become smaller or empty. If T is in a ready state the set cannot increase. Instead of immediately committing, or delaying the commitment for a given amount of time (which may results in aborts) the TTS can block the commitment of

15   T until all transactions in $ABORT_{co}(T)$ are decided. However, if another RM in the environment also blocks, this may result in a global deadlock.)

Definition:

20   41. A CORTTS is a COTTS which generates histories that are both CO and recoverable. The CORTTS maintains an enhanced serializability graph, wr-USG:

         wr-USG(H) = (UT, C, C_{wr}) where

UT is the set of all undecided transactions in the history H; and

25

C is the set of edges between transactions in UT. There is a C edge from T_1 to T_2 if and only if T is only in non $\underline{wr}$ conflicts with T_1. $C_{\underline{wr}}$ is the set of edges between transactions in UT with $\underline{wr}$ conflicts as well. There is a $C_{\underline{wr}}$ edge from T_1 to T_2 if and

5   only if T_2 is in a $\underline{wr}$ conflict with T_1 (and possibly also in conflicts of other types). C and $C_{\underline{wr}}$ are disjoint. The set of transactions aborted as a result of committing T (to prevent future commitment-ordering violation) is defined as follows:

$$\text{ABORT}_{co}(T) = \{T' \mid T' \to T \text{ is in C or } C_{\underline{wr}}\}$$

10   The definition of ABORT_{co} (T) here has the same semantics as of the respective set for COTTS. The set of aborted transactions due to recoverability, as a result of aborting transaction T', is defined as follows:

$$\text{ABORT}_{REC}(T') = \{ T'' \mid T' \to T'' \text{ is in } C_{\underline{wr}} \text{ or } T''' \to t'' \text{ is in } C_{\underline{wr}}$$

15   where T''' is in $\text{ABORT}_{REC}(T;)\}$. Note that the definition is recursive. This well reflects the nature of cascading aborts. The CORTTS iterates the following steps: (a) select any $\underline{ready}$ transaction T in $\underline{wr}$-USG, that does not have any in-coming $C_{\underline{wr}}$ edge (i.e. such that T is not in $\text{ABORT}_{REC}(T')$ for any transaction

20   T' in $\text{ABORT}_{co}(T)$; this to avoid a need to later abort T itself), and commit it; (b) abort all the transaction T' (both $\underline{ready}$ and $\underline{active}$) in $\text{ABORT}_{co}(T)$; (c) abort all the transaction T'' (both $\underline{ready}$ and $\underline{active}$) in $\text{ABORT}_{REC}(T')$ for every T' aborted in the previous step (cascading aborts); and (d) remove any $\underline{decided}$

25   transaction (T and all the aborted transactions) from the graph.

(Note: during each iteration wr-USG should reflect all operation's conflicts till commit.)

5   Theorem

42. The CORTTS produces CO, recoverable histories

Proof: The histories generated are CO by theorem 40 since CORTTS differs from COTTS only in aborting additional transactions during each iteration (due to the recoverability

10   requirement). Since all the transactions that can violate recoverability (transactions in $ABORT_{REC}(T')$ for every aborted transaction T' in $ABORT_{CO}(T)$ are aborted during each iteration (i.e. transactions that read data written by an aborted transaction before the abort), the generated histories are

15   recoverable. (Note: The CORTTS can be implemented as a non-blocking without delays, with delays and blocking TTS with results similar to those discussed above in the notes about the COTTS.)

20   Corollaries:

43. A COTTS generates histories that are serializable.

44. A CORTTS generates histories that are both serializable and recoverable

25

45. Non-blocking schedulers based on COTTS and CORTTS generate deadlock free executions only. (Note: The TTSs above can be combined with any resource access schedules (RAS) for scheduling resource access operations. If both the TTS and the RAS are non-

5   blocking, then also the combined mechanism is non-blocking, and hence ensures deadlock-freeness. A combination of a RAS and a TTS can replace a RAS above, if some filtering (by a TTS) is required to impose further history properties. In this case the filtering TTS may only abort transaction. But it does not matter

10 if the RAS produces serializable histories since the CO TTSs above guarantee serializability. The combined mechanism may execute as follows: First, the transactions are controlled by the RAS (or a RAS with a TTS). The unaborted, ready transactions are considered by the COTTS as candidates to be committed, and

15 transactions are aborted if violating the CO TTS's conditions. Note that if the scheduler above is S-S2PL based, then the USG of the respective CO TTS does not have any edges. This means that no aborts by the CO TTS are needed, as one can expect, and a CO TTS is unnecessary. This is an extreme case. Other scheduler

20 types can induce other properties of the respective USGs to impose desired schedule patterns and system behaviors, according to the nature of the transactions involved. Note also that if the combined CC mechanism guarantees the Recoverability property, the COTTS is sufficient (no need for CORTTS because

25 recoverability is inherited). When the scheduler if Timestamp Ordering (TO) based and CO is desired, an advantage can be taken

of existing data structures, rather than implementing the USG

independently. In this case CO is imposed by enforcing the

Timestamp Commitment Ordering rule.

5   Definitions:

46. An <u>environment</u> comprises a distributed services system

having a plurality of Resource Managers (RMs), where a

transaction can span any subset of <u>participating</u> RMs. Each RM in

an environment has an <u>identifier</u> (e.g. RM 2). Events are

10 qualified by both a transaction's identifier and an RM's

identifier (e.g. $w_{3,2}[x]$ means a write operation of data item x by

RM 2 on behalf of transaction T_3).

Axiom:

15 47. If $P_{i,j}[x]$, $q_{k,l}[y]$, $j \neq l$ are operations (by RMs j, l

respectively), then $x \neq y$; i.e. these operations cannot conflict.

Definitions:

48. A global transaction T_i consists of one or more <u>local</u>

20 <u>subtransactions</u>. A local subtransaction $T_{i,j}$, accesses <u>all</u> the

data under the control of RM j that T_i needs to access, and <u>only</u>

these data items (i.e. all its events are qualified with j). A

local subtransaction obeys the definition of a transaction. A

local subtransaction has the <u>states</u> of a transaction.

25

49. A <u>local history</u> is generated by a single RM, and defined over the set of its local subtransactions. A local history obeys the definition of a history in section 2. H_i is the history generated by RM i with a relation $<_{Hi}$. (Note: It is assumed

5 that an <u>atomic commitment</u> (AC) protocol is applied to guarantee <u>atomicity</u> in the distributed environment.)

50. An AC protocol implements the following general scheme each time a transaction is decided: Each participating RM votes

10 either "Yes" or "No" (but not both) after the respective local subtransaction has reached the "ready" state, or votes "No" if unable to reach the "ready" state. The transaction is committed by all RMs if and only if all have voted "Yes". Otherwise it is aborted by all the RMs. (Notes: 2PC is a special case of AC.

15 Failure and recovery issues are not dealt with here.) The fact that AC is used allows one to assume that a distributed transaction has a single commit event (though in reality this is not always guaranteed). However, this is not true for aborts.

Example: The following two transactions both access data

20 items x and y. x, y are under the control of RMs 1, 2 respectively. T_1 and T_2 and their local transactions are the following:

RM 1 $T_{1,1}$: $r_{1,1}[x]$ c_1 $T_{2,1}$: $w_{2,1}[x]$ c_2

RM 2 $T_{1,2}$: $r_{1,2}[y]$ c_1 $T_{2,2}$: $w_{2,2}[y]$ c_2

25   T_1 T_2

The RMs generate the following (local) histories H_1 and H_2:

RM 1: H_1 $r_{1,1}[x]$ $w_{2,1}[x]$ c_2 c_1

RM 2: H_2 $w_{2,2}[y]$ c_2 $r_{1,2}[y]$ c_1

Note that the history H_1 violates commitment-ordering which results in a (global) serializability violation. The respective

5 global history H is described by the following order relationships:

$r_{1,1}[x]$ < $w_{2,1}[x]$ < c_2 < $r_{1,2}[y]$ < c_1

$w_{2,2}[y]$ < c_2

10 51. For any history property X a (global) history H is in Local-X (is <u>locally</u> X) if and only if for every RM i in the environment H_i (the history of RM i) is in X (is X).

Theorem:

15 52. A history is in X (is globally X) if and only if it is in Local-X (i.e. Local-X = X), where X is any of the following property: REC, ACA, ST, CO, S-S2PL.

Proof: Follows from the definition of Local-X, Axiom 47, and the definitions of REC, ACA ST, CO and S-S2PL.

20

Theorem:

53. Being in Local-X does not imply that a history is in X (i.e. Local-X ⊃ X), where X is any of the following properties: SER, 2PL, S2PL

25 Proof: Let H be the history as in the example above. The history H is in Local-SER, Local-2PL and Local-S2PL since both H_1

and H_2 are in SER, 2PL and S2PL. However H is not in SER, 2PL or

S2PL. CSC(H) has a cycle, so by the H is not in SER. If it is

in 2PL it is also in SER, and we have a contradiction.

5   Theorem:

54.   SER ⊃ Local-CO.   In other words if a history is in Local-CO

then it is globally serializable. This theorem follows from the

Commitment Ordering theorem and theorem 52.

10   (Note: Local-CO can be maintained by RMs using any types of CO

mechanisms.)

Definition:

15   55. A permanent risk (PR) transaction is a transaction that can

cause a potential serializability violation when committed, and

will stay in this situation forever.   The PR property is

relative to the resource manager. The above requirement implies

that each RM in the environment should implement the following

20   Commitment Strategy (CS): Starting from a history with no

decided transactions any ready transaction is committed (usually

the RM is requested via an AC protocol to commit some

transaction). Every other transaction that is a PR is aborted.

(hidden axiom is assumed here, that resources are not held

25   unnecessarily. Otherwise PR transactions can be marked and kept

undecided for ever.) Then another (any) ready transaction, that

cannot cause a serializability violation, is committed. Again all the PR transactions are aborted, and so forth.

Theorem:

5   56. If only local serializability information is available for each RM in the environment, and if atomic commitment is applied, then CS is a necessary strategy for each RM, in order to guarantee global serializability. CS produces locally CO histories (global histories in Local-CO).

10   Proof: The Serializability Theorem implies that the serializability graph provides all the necessary information about serializability. We assume that every RM, say RM i, "knows" its local serializability graph SG_i (it includes all the committed and undecided transactions only) and its subgraphs CSG_i

15   (includes committed transactions only) and USG_i (includes all undecided transactions). We also assume (based on AC) that each RM has committed a transaction, if and only if it has voted "Yes", and "knows" that all other RMs participating in a transaction have voted "Yes", and will eventually commit it. The

20   goal for each RM is to guarantee a cycle-free (global) CSG (committed transaction serializability graph), by avoiding any action that may create a global cycle (local cycles in CSG_i are eliminated by RM i). First, CS is trivially necessary for the following reasons: Since a PR transaction remains PR for ever

25   (by definition), it cannot be committed and must be aborted to free resources. On the other hand, any ready transaction that

cannot cause a serializability violation can be committed. We now need to identify permanent risk (PR) transactions, while implementing CS. We show that this implies that each RM operates as a COTTS. Each RM implements CS as follows:

5   (a) Base stage: Assume that CSG_i does not include any transaction. Commit any ready transaction T. Suppose that prior to committing T there is an edge $T' \to T$ in USG_i. It is possible that there is an edge $T \to T'$ in some USG_j of some RM j, $j \neq i$, but RM i, though, cannot verify this. This means that committing T'

10   later may cause a cycle in CSG. Since committing T cannot be reversed (see transaction state transitions in section 3), no event can change this situation. Hence T' is a PR, and RM i must abort it.

     (b) Inductive stage: Suppose that CSG_i includes one

15   transaction at least. We show that no ready transaction can cause a serializability violation if committed, and hence can be committed (provided that a consensus to commit is reached by all the participating RMs via AC): Commit any ready transaction T. Examine any previously committed transaction T". It is

20   impossible to have a path $T \to \dots \to T"$ in CSG_i or in CSG_j for any RM j, $j \neq i$ since if this path existed at the stage when T" was committed, it would have been disconnected during that stage, when aborting all transactions with edges to T" (using the arguments given for the base stage above), and since no incoming

25   edges to T" could have been generated after T" has been committed. Hence, only a path $T" \to \dots \to T$ can exist in CSG_i or

in CSG_j for any RM j, $j \neq i$. This means that no cycle in CSG through T and T" can be created, and no T" needs to be aborted (which would fail the strategy). Examine now any undecided transaction T' (in USG_i.) Suppose that prior to committing T

5   there is an edge T' → T in USG_i. Using again the arguments given for the base stage, T' is a PR, and RM i must abort it (by voting "No" via AC). If there is no edge from T' to T, no decision concerning T' is taken at this stage.

The arguments above ensure that no ready transaction can

10   cause a serializability violation when committed at the beginning of an inductive stage, as was assumed, and hence (any ready transaction) T could have been committed.

In the CS implementation above all the PR transactions are identified and aborted at each stage. Examining this

15   implementation we conclude that it functions as a COTTS. Hence, by theorem 40 every RM involved produces a CO history, and the generated (global) history is locally CO (in Local-CO). The only possible deviation from the implementation above is by aborting additional transactions at each stage. Such a deviation still

20   maintains the generated history in Local-CO.

Corollary
57. If RMs coordinate concurrency control via <u>atomic commitment only</u>, then <u>local commitment-ordering</u> is a <u>necessary</u> and

25   <u>sufficient condition</u> for (global) <u>serializability</u>. This corollary follows from theorems 52, 55 and 56.

Corollary

58. If RMs coordinate concurrency control via <u>atomic commitment</u> only, then <u>local commitment-ordering</u> and <u>local recoverability</u> are a <u>necessary</u> and <u>sufficient condition</u> for (global) <u>serializability</u> and <u>recoverability</u>. This corollary follows from Theorem 52.

59. A <u>global deadlock</u> is a deadlock caused by a mutual blocking of two or more <u>local subtransactions</u> in two different transactions at least, by two different RMs at least. (Notes: Since commitment-ordering is not inherently blocking, it can be implemented in a non blocking manner, for example, by aborts or by aborts after delays. If the schedulers of all the RMs in the environment are non-blocking (with the exception of one that can be blocking), the executions are <u>deadlock-free</u>.

Another way to implement commitment-ordering is by using blocking CO certifiers (CO schedulers with a non-blocking RAS and a blocking TTS). If the schedulers for all RMs are certifiers, when two at least are blocking, the global deadlocks can occur (even if each RM resolves its own deadlocks). In this case all the transactions involved with a deadlock are in a ready state. This fact allows to resolve deadlocks during atomic commitment.

If schedulers of two or more RMs are blocking, where one, at least, has a blocking RAS (e.g. S-S2PL or CO, BTO based), then also active transactions can be involved with a global deadlock. In this case atomic commitment messages are not sufficient for deadlock resolution, and additional messages signaling the

presence of blocks (possibly piggy-backed on AC messages of other transactions) are required).

Conclusion

5   Commitment-ordering provides a way to achieve global serializability, also through deadlock free mechanisms. This allows a trade off between blocking techniques, that are subject to deadlocks, and non-blocking implementations, that are deadlock free but subject to cascading aborts. In order to guarantee

10 serializability no services, but those of atomic commitment, are necessary for the coordination of transaction management across RMs, if each RM supports commitment-ordering. However, commitment-ordering is necessary for global serializability if only atomic commitment is used for RM coordination.

WHAT IS CLAIMED IS:

1. In a multiprocessor computing system in which a global co-ordinator communicates with a plurality of transaction processors by way of "prepare" and "commit" commands, a method of operating a digital processor to process transactions which might have conflicting component opertions, said method comprising the steps of:

a) beginning preparation of results of said transactions;

b) determining a commitment order for said transactions when one of said transactions has a first operation that conflicts with a second operation in another one of said transactions, said conflicting operations having an order of performance, said commitment order being the same as said order of performance;

c) committing to memory state of said computing system prepared results of a selected one of said transactions;

d) aborting an abort set of said transactions for which commitment is contrary to said commitment order and the committing of said selected one of said transactions.

2. The method as claimed in claim 1, wherein said step of committing to memory state includes updating a data base file.

3. The method as claimed in claim 1, wherein said step of committing to memory state includes changing memory state of a data object.

4. The method as claimed in claim 1, further comprising the step of selecting said selected one of said transactions by comparing predetermined priorities assigned to said transactions.

5. 5. The method as claimed in claim 1, further comprising the step of selecting said selected one of said transactions based on ordering of said transactions in a list.

6. The method as claimed in claim 1, further comprising the

10 step of selecting said selected one of said transactions in response to a commit command form a coordinator.

7. The method as claimed in claim 1, further comprising the step of selecting said selected one of said transactions in order

15 to minimize the number of transactions that are aborted in said step d).

8. The method as claimed in claim 1, further comprising the step of receiving from a coordinator a request to prepare a

20 specified one of said transactions, and delaying acknowledgement of completion of preparation of said specified one of said transactions until none of said transactions not yet committed in step c) nor aborted in step d) are contrary to said predefined commitment order and the committing of said specified one of said

25 transactions.

9.   The method as claimed in claim 8, further comprising the step of terminating said delaying when said delaying persists for a predetermined duration of time.

5   10.   The method as claimed in claim 8, further comprising the step of terminating said delaying upon receipt of a termination signal from a coordinator.

11.   The method as claimed in claim 1, wherein a read
10   operation of a second one of said transactions reads write data written by a write operation of a first one of said transactions before said first one of said transactions is committed, and wherein said method further comprises the step of aborting all of said transactions that have read data written by aborted
15   transactions.

12.   A method of operating a digital computer to process transactions in a computing system, said method comprising the steps of:
20   a) receiving requests for processing said transactions;
        b) beginning preparation of results of said transactions;
        c) determining a commitment order for said transactions when one of said transactions has a first operation that conflicts with a second operation in another one of said transactions, said
25   conflicting operations having an order of performance, said commitment order being the same as said order of performance,

said commitment order being determined during the preparation of results for transactions having conflicting operations;

    d) storing in memory data defining said commitment order;

    e) committing to memory state of said computing system

5   prepared results of a selected one of said transactions;

    f) inspecting said data stored in memory to determine whether commitment for other of said transactions is contrary to the committing of said selected one of said transactions and said commitment order for said transactions, and when commitment for

10   other of said transactions is contrary to the committing of said selected one of said transactions and said commitment order for said transactions, aborting said other of said transactions for which commitment is contrary to said commitment order and the committing of said selected one of said transactions.

15

    13. The method as claimed in claim 15, wherein said data is stored in said memory in the form of a directed graph, and wherein said method further comprises the step of removing from said graph data with respect to committed and aborted

20   transactions.

    14. The method as claimed in claim 12, further comprising the step of receiving from a coordinator a request to prepare a specified one of said transactions, and delaying acknowledgement

25   of completion of preparation of said specified one of said transactions until none of said transactions not yet committed in

step d) nor aborted in step e) are contrary to said predefined

commitment order and the committing of said specified one of said

transactions.

5   15. A method of operating a digital computer to process

transactions in a computing system, said method comprising the

steps of:

    a) receiving requests to perform transactions;

    b) scheduling performance of operations of said transactions

10 on a real-time basis such that operations of some transactions

are performed in accordance with availability of resources of

said digital computer before commitment of other transactions;

    c) determining a commitment order for said transactions when

one of said transactions has a first operation that conflicts

15 with a second operation in another one of said transactions, said

conflicting operations having an order of performance, said

commitment order being the same as said order of performance; and

    d) enforcing commitment of selected ones of said

transactions in accordance with said commitment order.

20

    16. In a multiprocessor computing system in which a co-

ordinator communicates with a plurality of transaction processors

by way of "prepare" and "commit" comands, a digital computer

system for processing transactions, which might have conflicting

25 component operations said digital computer system comprising, in

combination:

a) means for scheduling performance of operations of said transactions on a real-time basis such that operations of some transactions are performed in accordance with availability of resources of said digital computer system before commitment of

5   other transactions;

b) means for determining a commitment order for said transactions when one of said transactions has a first operation that conflicts with a second operation in another one of said transactions, said conflicting operations having an order of

10   performance, said commitment order being the same as said order of performance; and

c) means for enforcing commitment of selected ones of said transactions in accordance with said commitment order, said means for enforcing including means for delaying commitment of selected

15   transactions and means for aborting an abort set of said transactions for which commitment is contrary to said commitment order and the committing of said selected transactions.

17.   The digital computer system as claimed in claim 16,

20   wherein said means for delaying includes means for aborting selected transactions after a delay of a predetermined duration of time.

18.   The digital computer system as claimed in claim 38,

25   wherein said means for delaying includes means for delaying acknowledgement of completion of preparation of said selected

transactions until none of said transactions that are not yet
committed nor aborted are contrary to said predefined commitment
order and committing of said requested one of said transactions.

5   19. The digital computer system as claimed in claim 16,
wherein said means for determining said commitment order includes
means for detecting the performance of an operation that
conflicts with an operation previously performed.

10   20. The digital computer system as claimed in claim 19,
wherein said means for detecting includes means for comparing an
address of a memory access operation for one transaction to
addresses of memory access operations previously performed for
other transactions.

15

21. The digital computer system as claimed in claim 16,
wherein said means for aborting includes means for aborting all
of said transactions that have read data written by aborted
transactions.

20

Patents Act 1977
Examiner's report to the Comptroller under Section 17 (The Search Report)

−88−

Application number
9210876.0

Relevant Technical fields	Search Examiner
(i) UK Cl (Edition K) G4A (AMB, AUD)	B G WESTERN
(ii) Int Cl (Edition 5) G06F (9/46, 15/16)	

Databases (see over)	Date of Search
(i) UK Patent Office	
(ii)	6 AUGUST 1992

Documents considered relevant following a search in respect of claims   1−21

Category (see over)	Identity of document and relevant passages	Relevant to claim(s)
A	EP-0457112 A2 (IBM) Whole document	1−21
A	US 4249241 A (ABERLE ET AL) Whole document	1−21
A	US 4881166 A (THOMPSON ET AL) Whole document	1−21

Category	Identity of document and relevant passages	Relevant to claim(s)

Categories of documents

X: Document indicating lack of novelty or of inventive step.

Y: Document indicating lack of inventive step if combined with one or more other documents of the same category.

A: Document indicating technological background and/or state of the art.

P: Document published on or after the declared priority date but before the filing date of the present application.

E: Patent document published on or after, but with priority date earlier than, the filing date of the present application.

&: Member of the same patent family, corresponding document.

Databases: The UK Patent Office database comprises classified collections of GB, EP, WO and US patent specifications as outlined periodically in the Official Journal (Patents). The on-line databases considered for search are also listed periodically in the Official Journal (Patents).

THIS PAGE BLANK (USPTO)